Orientação a Objetos e SOLID para Ninjas

Projetando classes flexíveis

Casa do Código

SÉRIE CAELUM

MAURÍCIO ANICHE

© Casa do Código
Todos os direitos reservados e protegidos pela Lei nº9.610, de 10/02/1998.
Nenhuma parte deste livro poderá ser reproduzida, nem transmitida, sem autorização prévia por escrito da editora, sejam quais forem os meios: fotográficos, eletrônicos, mecânicos, gravação ou quaisquer outros.

Casa do Código
Livros para o programador
Rua Vergueiro, 3185 - 8º andar
04101-300 – Vila Mariana – São Paulo – SP – Brasil

Agradecimentos

Agradecer é sempre a parte mais difícil. Em primeiro lugar, agradeço a meus pais por sempre apoiarem toda e qualquer iniciativa que tenho. Ver o quanto eles ficam orgulhosos a cada pequena conquista em minha carreira me faz querer sempre mais e mais.

Agradeço também ao meu amigo e colega de trabalho, Guilherme Silveira, pelas incontáveis discussões técnicas que temos por e-mail, pessoalmente, telefone, gtalk, e todos os outros meios de comunicação possíveis. Suas opiniões fortes e bem decididas sobre diversos pontos de engenharia de software sempre me fazem pensar a respeito das minhas opiniões. É sempre bom ter pessoas prontas para debater qualquer assunto técnico.

Não posso deixar de agradecer também ao meu grupo de pesquisa na USP, que de maneira indireta sempre me faz aprender muito sobre engenharia de software. Fica então meu abraço ao Marco Gerosa, Gustavo Oliva, Igor Steinmacher e Igor Wiese.

E, em um livro sobre orientação a objetos, não posso deixar de agradecer ao meu professor da graduação, Ismar Frango Silveira, que realmente me ensinou o que é orientação a objetos. Sua paixão pelo ensino também me motivou muito a me tornar professor e passar conhecimento adiante.

Agradeço também ao Rodrigo Turini, excelente desenvolvedor da Caelum, que leu o livro, e me deu excelentes feedbacks e sugestões de melhoria.

Por fim, a todos meus amigos, que me aguentam falando o tempo todo sobre o quanto é difícil e trabalhoso escrever um livro.

Forte abraço a todos!

Quem sou eu?

Meu nome é Mauricio Aniche e trabalho com desenvolvimento de software há por volta de 10 anos. Em boa parte desse tempo, atuei como consultor para diferentes empresas do mercado brasileiro e internacional. Com certeza, as linguagens mais utilizadas por mim ao longo da minha carreira foram Java, C# e C.

Como sempre pulei de projeto em projeto (e, por consequência, de tecnologia em tecnologia), nunca fui a fundo em nenhuma delas. Pelo contrário, sempre foquei em entender princípios que pudessem ser levados de uma para outra, para que no fim, o código saísse com qualidade, independente da tecnologia.

Justamente por esse motivo fui obrigado a pesquisar por melhores práticas de programação e como escrever código bom em todo lugar e projeto de que participava. Não teve jeito: acabei caindo em orientação a objetos. E de lá pra cá, venho estudando com muito afinco boas práticas de programação.

Já participei de diversos eventos da indústria e academia brasileira, como é o caso da Agile Brazil, WBMA, SBES, e internacionais, como o ICSME, ICST, PLoP e Agile. Em 2007, concluí meu mestrado pela Universidade de São Paulo, onde pesquisei sobre TDD e seus efeitos no projeto de classes. Atualmente, trabalho pela Caelum, como consultor e instrutor. Hoje faço doutorado na Universidade de São Paulo, onde pesquiso sobre métricas e evolução de software.

Você pode ler mais sobre meu trabalho em meu blog: www.aniche.com.br.

Sumário

1 Orientação a Objetos, pra que te quero? 1

1.1 Qual o público deste livro? 3

2 A coesão e o tal do SRP 5

2.1 Um exemplo de classe não coesa 6
2.2 Qual o problema dela? 7
2.3 Em busca da coesão 7
2.4 Uma pitada de encapsulamento 10
2.5 Quando usar métodos privados? 13
2.6 Falta de coesão em controllers 15
2.7 Inveja da outra classe 19
2.8 SRP Single Responsibility Principle 19
2.9 Separação do modelo, infraestrutura, e a tal da arquitetura hexagonal 21
2.10 Conclusão 22

3 Acoplamento e o tal do DIP 23

3.1 Qual o problema dela? 24
3.2 Estabilidade de classes 26
3.3 Buscando por classes estáveis 27
3.4 DIP Dependency Inversion Principle 31
3.5 Um outro exemplo de acoplamento 33
3.6 Dependências lógicas 36
3.7 Conclusão 37

4 Classes abertas e o tal do OCP **39**
4.1 Qual o problema dela? 41
4.2 OCP Princípio do Aberto-Fechado 43
4.3 Classes extensíveis 46
4.4 A testabilidade agradece! 47
4.5 Um exemplo real 50
4.6 Ensinando abstrações desde a base 55
4.7 Conclusão 56

5 O encapsulamento e a propagação de mudanças **57**
5.1 Qual o problema dela? 58
5.2 Intimidade inapropriada 60
5.3 Um sistema OO é um quebra-cabeças 61
5.4 Tell, Don't Ask 63
5.5 Procurando por encapsulamentos problemáticos 64
5.6 A famosa Lei de Demeter 66
5.7 Getters e setters pra tudo, não! 67
5.8 Corrigindo o código inicial 69
5.9 Modelos anêmicos 71
5.10 Conclusão 73

6 Herança x composição e o tal do LSP **75**
6.1 Qual o problema dela? 77
6.2 LSP Liskov Substitutive Principle 77
6.3 O exemplo do quadrado e retângulo 78
6.4 Acoplamento entre a classe pai e a classe filho 80
6.5 Favoreça a composição 83
6.6 Herança para DSLs e afins 85
6.7 Quando usar herança então? 87
6.8 Pacotes: como usá-los? 88
6.9 Conclusão 88

7 Interfaces magras e o tal do ISP **89**
7.1 Interfaces coesas e magras 91
7.2 Pensando na interface mais magra possível 92
7.3 E os tais dos repositórios do DDD? 95
7.4 Fábricas ou injeção de dependência? 95
7.5 Conclusão 98

8 Consistência, objetinhos e objetões **99**
8.1 Construtores ricos 100
8.2 Validando dados 103
8.3 Teorema do bom vizinho e nulos para lá e para cá 108
8.4 Tiny Types é uma boa ideia? 109
8.5 DTOs do bem 111
8.6 Imutabilidade x mutabilidade 113
8.7 Classes que são feias por natureza 115
8.8 Nomenclatura de métodos e variáveis 116
8.9 Conclusão 117

9 Maus cheiros de design **119**
9.1 Refused Bequest 120
9.2 Feature Envy 121
9.3 Intimidade inapropriada 122
9.4 God Class 123
9.5 Divergent Changes 123
9.6 Shotgun Surgery 124
9.7 Entre outros 125

10 Métricas de código **127**
10.1 Complexidade ciclomática 128
10.2 Tamanho de métodos 129
10.3 Coesão e a LCOM 130
10.4 Acoplamento aferente e eferente 132
10.5 Má nomenclatura 134
10.6 Como avaliar os números encontrados? 134
10.7 Ferramentas 135

11 Exemplo prático: MetricMiner **137**
11.1 O projeto de classes do ponto de vista do usuário 138
11.2 O projeto de classes do ponto de vista do desenvolvedor . . . 141
11.3 Conclusão 144

12 Conclusão **147**
12.1 Onde posso ler mais sobre isso? 147
12.2 Obrigado! 148

Capítulo 1

Orientação a Objetos, pra que te quero?

O termo **orientação a objetos** não é mais muita novidade para ninguém. Todo curso de programação, inclusive os introdutórios, já falam sobre o assunto. Os alunos saem de lá sabendo o que são classes, a usar o mecanismo de herança e viram exemplos de Cachorro, Gato e Papagaio para entender polimorfismo. O que ainda é novidade é ver todas essas coisas aplicadas em projetos do mundo real. Usar OO é muito mais difícil do que parece e, na prática, vemos código procedural disfarçado de orientado a objeto.

A diferença entre código procedural e orientado a objetos é bem simples. Em códigos procedurais, a implementação é o que importa. O desenvolvedor pensa o tempo todo em escrever o melhor algoritmo para aquele problema, e isso é a parte mais importante para ele. Já em linguagens orientadas a objeto,

a implementação também é fundamental, mas pensar no projeto de classes, em como elas se encaixam e como elas serão estendidas é o que importa.

Pensar em um sistema orientado a objetos é, portanto, mais do que pensar em código. É desenhar cada peça de um quebra-cabeça e pensar em como todas elas se encaixarão juntas. Apesar de o desenho da peça ser importante, seu formato é ainda mais essencial: se você mudar o formato de uma peça, essa mudança precisará ser propagada para as peças ao redor, que mudarão e propagarão essa mudança para as outras peças ao redor. Se você pensar em cada classe como uma peça do quebra-cabeça, verá que o problema ainda é o mesmo. Precisamos pensar em como fazer classes se encaixarem para trabalharem juntas. O desenho da peça é importante, mas se um deles estiver cheio, é mais fácil jogar fora e fazer uma nova peça com o mesmo formato e um desenho novo, do que mudar o formato.

A parte boa é que desenhar o formato do quebra-cabeça é muitas vezes mais legal do que fazer o desenho de cada uma delas. A parte chata é que isso é mais difícil do que parece.

Se você já trabalhou em um sistema orientado a objetos, conhece alguns dos problemas que eles podem apresentar. Projetos de classes são difíceis de mudar e tudo parece um tapete de dominós, onde uma mudança é propagada imediatamente para a próxima classe. Ou mesmo projetos frágeis, nos quais uma mudança em um ponto específico do sistema quebra muitos outros pontos. E quanto ao reúso? Muitos módulos são impossíveis de serem reutilizados e só servem para aquele ponto específico em que foi criado, forçando o desenvolvedor a repetir código para lá e para cá.

Realmente, muita coisa pode dar errado em um sistema orientado a objetos. Meu objetivo neste livro é justamente ajudar você a fazer projetos de classe que evitem esses problemas. Tenho certeza de que você já sabe o que é acoplamento, coesão e encapsulamento. Aqui vou mostrar um outro ponto de vista para cada uma dessas palavras tão importantes. Mostrarei a importância de termos abstrações e como criar pontos de flexibilização, de maneira a facilitar a evolução e manutenção de nossos sistemas.

Como é praticamente impossível (para não dizer, inútil) discutir OO sem mostrar pequenos trechos de código, todos os capítulos os contêm. Mas lembre-se que eles são contextuais e servem basicamente de motivação para

as discussões. Generalize cada argumento feito. Entenda o essencial por trás de cada um deles.

1.1 Qual o público deste livro?

Este livro não é um guia básico de orientação de objetos. Pelo contrário, discuto aqui conceitos avançados sobre os grandes pilares da OO. Idealmente, o leitor já trabalha em alguma linguagem orientada a objetos e tem um pouco de teoria sobre o assunto.

Para esse leitor, o livro, sem dúvida, dará uma nova visão sobre projeto de classes. Espero, de coração, que o livro o ajude a aprender e a escrever softwares ainda melhores.

Capítulo 2

A coesão e o tal do SRP

Coesão é, com certeza, uma das palavras mais conhecidas por programadores que usam linguagens orientadas a objeto. Seu significado também é bastante conhecido: **uma classe coesa é aquela que possui uma única responsabilidade**. Ou seja, ela não toma conta de mais de um conceito no sistema. Se a classe é responsável por representar uma Nota Fiscal, ela representa apenas isso. As responsabilidades de uma Fatura, por exemplo, estarão em outra classe.

Classes coesas são vitais em um sistema orientado a objetos. Elas são mais simples de serem mantidas, possuem menos código e seu reúso é maior. Mas a pergunta é: **como escrever classes coesas?**. Afinal, não é novidade para ninguém que por aí encontramos classes gigantes, com dezenas de métodos, dificílimas de serem mantidas.

Neste capítulo, discutirei um pouco sobre como lutar para que nossas classes sejam sempre coesas; e, se tudo der certo, ganhar a maioria dessas

lutas.

2.1 Um exemplo de classe não coesa

Suponha uma classe `CalculadoraDeSalario`. Ela, como o próprio nome já diz, é responsável por calcular salários do funcionário. A regra é razoavelmente simples (até por questões didáticas): de acordo com o cargo e o salário dele, o desconto aplicado é diferente. Observe isso implementado no trecho de código a seguir:

```
class CalculadoraDeSalario {
    public double calcula(Funcionario funcionario) {
        if(DESENVOLVEDOR.equals(funcionario.getCargo())) {
            return dezOuVintePorcento(funcionario);
        }

        if(DBA.equals(funcionario.getCargo()) ||
         TESTER.equals(funcionario.getCargo())) {
            return quinzeOuVinteCincoPorcento(funcionario);
        }

        throw new RuntimeException("funcionario invalido");
    }
}
```

Repare que cada uma das regras é implementada por um método privado, como o `dezOuVintePorcento()` e o `quinzeOuVinteCincoPorcento()`. A implementação desses métodos poderia ser como o seguinte:

```
private double dezOuVintePorcento(Funcionario funcionario) {
    if(funcionario.getSalarioBase() > 3000.0) {
        return funcionario.getSalarioBase() * 0.8;
    }
    else {
        return funcionario.getSalarioBase() * 0.9;
    }
}
```

Já escreveu algo parecido?

2.2 Qual o problema dela?

Tente generalizar esse exemplo de código. Códigos como esse são bastante comuns: é normal olhar para uma característica do objeto e, de acordo com ela, tomar alguma decisão. Repare que existem apenas 3 cargos diferentes (desenvolvedor, DBA e tester) com regras similares. Mas em um sistema real, essa quantidade seria grande. Ou seja, essa classe tem tudo para ser uma daquelas classes gigantescas, cheias de *if* e *else*, com que estamos acostumados. **Ela não tem nada de coesa**.

Imagine só essa classe com 15, 20, 30 cargos diferentes. A sequência de `if`s seria um pesadelo. Além disso, cada cargo teria sua implementação de cálculo diferente, ou seja, mais algumas dezenas de métodos privados. Agora tente complicar um pouco essas regras de cálculo. Um caos.

Classes não coesas ainda têm outro problema: a chance de terem defeitos é enorme. Talvez nesse exemplo seja difícil de entender o argumento, mas como essas muitas regras estão uma perto da outra, é fácil fazer com que uma regra influencie a outra, ou que um defeito em uma seja propagado para a outra. Entender e manter esse arquivo não é uma tarefa fácil ou divertida de ser feita.

Agora imagine que exista a necessidade de se reutilizar o método `dezOuVintePorcento()` em algum outro ponto do sistema. Levar a classe `CalculadoraDeSalario` inteira para outro sistema, ou mesmo fazer outra classe depender dela só para reutilizar esse comportamento, é um pecado. Reutilizar código está difícil também.

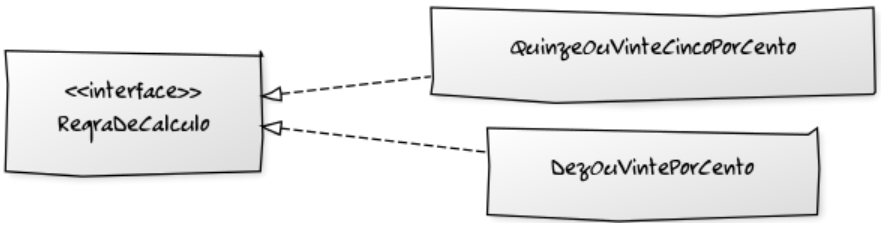
A pergunta é: como resolver esses problemas?

2.3 Em busca da coesão

O primeiro passo para tal é entender por que essa classe não é coesa. Uma maneira é olhar para a classe e descobrir o que faria o desenvolvedor escrever mais código nela. Repare, **toda classe que é não coesa não para de crescer nunca**.

A classe `CalculadoraDeSalario`, em particular, cresce indefinidamente por dois motivos: sempre que um cargo novo surgir ou sempre que uma regra de cálculo nova surgir. Vamos começar resolvendo o segundo problema, já que ele é mais fácil.

Observe cada método privado. Apesar de terem implementações diferentes, eles possuem o mesmo "esqueleto" (ou seja, forma, abstração). Ambos recebem um funcionário e nos devolvem um `double` com o salário calculado. A ideia, portanto, será colocar cada uma dessas regras em classes diferentes, todas implementando a mesma interface.



Se implementarmos dessa forma, repare que cada regra de cálculo agora está bem isolada, isto é, será bem difícil que uma mudança em uma das regras afete a outra regra. Cada classe contém apenas uma regra, fazendo essa classe muito coesa, afinal ela só mudará se aquela regra em particular mudar. Cada classe também terá pouco código (o código necessário para uma regra de cálculo, claro), muito melhor do que a antiga classe grande.

Em código, isso é simples. Veja a interface que representa a regra de cálculo:

```
public interface RegraDeCalculo {
    double calcula(Funcionario f);
}
```

As implementações das regras são idênticas às implementações dos métodos privados, com a diferença de que agora cada uma está em sua respectiva classe:

```
public class DezOuVintePorCento implements RegraDeCalculo {
    public double calcula(Funcionario funcionario) {
        if(funcionario.getSalarioBase() > 3000.0) {
            return funcionario.getSalarioBase() * 0.8;
        }
        else {
            return funcionario.getSalarioBase() * 0.9;
        }
    }
}

public class QuinzeOuVinteCincoPorCento implements
RegraDeCalculo {
    public double calcula(Funcionario funcionario) {
        if(funcionario.getSalarioBase() > 2000.0) {
            return funcionario.getSalarioBase() * 0.75;
        }
        else {
            return funcionario.getSalarioBase() * 0.85;
        }
    }
}
```

Decisão de design tomada. Toda nova regra deve ser colocada em uma classe separada. Esqueça agora a implementação e foque no que fizemos. A classe `CalculadoraDeSalario`, que era grande e complexa cresceria para sempre, pois não era coesa foi quebrada em várias classes menores. Responsabilidades separadas, classes menores, mais fáceis de serem mantidas, reutilizadas etc.

Precisamos agora resolver o próximo problema. Este, em particular, está menos ligado à coesão, mas ainda sim é digno de nota.

Podemos separar as responsabilidades em diferentes métodos?

Aqui optamos por dividir as responsabilidades em classes. A pergunta é: será que poderíamos separar em métodos? O problema é justamente o reúso. Se quebrarmos um método grande em vários pequenos métodos privados, não teremos reúso daquele comportamento isolado.

Métodos privados são excelentes para melhorar a legibilidade de um método maior. Se você percebeu que existem duas diferentes responsabilidades em uma mesma classe, separe-os de verdade.

2.4 Uma pitada de encapsulamento

Sempre que o desenvolver ouve "código ruim", ele logo imagina uma implementação complicada, cheia de `ifs` e variáveis com maus nomes. Sim, implementações ruins são problemáticas, mas quando se pensa em manter um sistema grande, a longo prazo, temos problemas maiores.

Um dos principais problemas em software é justamente a propagação de alterações. Quantas vezes você, ao por a mão em um sistema legado, precisou fazer a mesma alteração em pontos diferentes? E como você fez pra achar esses pontos de alteração? Se você apelou para algum tipo de `CTRL+F` ou `GREP`, você já passou pelo problema que estou discutindo. Em sistemas mal projetados, os pontos de alteração são implícitos. O desenvolvedor precisa buscá-los manualmente. E, óbvio, ele vai deixar passar algum.

Isso acontece em nosso código atual. Repare na implementação da classe `CalculadoraDeSalario` a seguir, que já faz uso das novas classes de regra de cálculo:

```
public double calcula(Funcionario funcionario) {
    if(DESENVOLVEDOR.equals(funcionario.getCargo())) {
       return new DezOuVintePorCento().calcula(funcionario);
    }

    if(DBA.equals(funcionario.getCargo()) ||
      TESTER.equals(funcionario.getCargo())) {
```

```
            return new QuinzeOuVinteCincoPorCento().calcula(
                            funcionario);
        }

        throw new RuntimeException("funcionario invalido");
    }
```

Nessa implementação, `DESENVOLVEDOR`, `DBA` e `TESTER` são *enums*. Sempre que um cargo novo surgir, o desenvolvedor é obrigado a adicionar um novo item nesse *enum* e alterar a classe `CalculadoraDeSalario` e fazê-la suportar esse novo cargo. Mas como sabemos disso? E se tivermos outras classes similares a essa calculadora?

Precisamos deixar essa decisão de design mais clara. O desenvolvedor deve saber rapidamente que, ao criar um cargo novo, uma regra de cálculo deve ser associada a ele. Precisamos **encapsular** melhor todo esse problema, para que a mudança, quando feita em um único ponto, seja propagada naturalmente.

Entendendo a motivação e o problema, é fácil ver que existem muitas implementações diferentes. Aqui, optarei por fazer uso da facilidade de *enums* no Java. O *enum* `Cargo` receberá no construtor a regra de cálculo. Dessa forma, qualquer novo cargo deverá, obrigatoriamente, passar uma regra de cálculo. Veja:

```
public enum Cargo {
    DESENVOLVEDOR(new DezOuVintePorCento()),
    DBA(new QuinzeOuVinteCincoPorCento()),
    TESTER(new QuinzeOuVinteCincoPorCento());

    private RegraDeCalculo regra;

    Cargo(RegraDeCalculo regra) {
        this.regra = regra;
    }

    public RegraDeCalculo getRegra() {
        return regra;
    }
}
```

Novamente, o principal aqui é entender o problema resolvido e não tanto a implementação. Se seu código exige que uma mudança seja feita em vários pontos diferentes para que ela seja propagada, talvez você esteja passando por um problema de projeto. Refatore seu código e encapsule esse comportamento em um único lugar. Lembre-se que o programador não deve nunca usar `CTRL+F` para programar e buscar pelos pontos de mudança.

Algumas mudanças precisam ser feitas em mais de um ponto

Algumas vezes não temos como não alterar em mais de um ponto. Por exemplo, sempre que você muda uma entidade no seu sistema, provavelmente essa mudança deverá ser propagada para um método do seu *controller*, e para uma `JSP` que representa sua *view*. Não há como fugir.

Mas esse tipo de mudança é menos problemático, afinal, essa é uma decisão explícita de design. Ao seguir o MVC, você sabe que a alteração em uma entidade pode ser propagada para o controller e para a view. O problema são as dependências implícitas; aquelas que você não sabe que existem.

MVC Model View Controller

MVC (*Model-View-Controller*) é um padrão arquitetural, criado há muito tempo, e que hoje é bastante popular. A ideia por trás do MVC é dividir a aplicação em três grandes partes: a camada de modelo, onde vivem as classes e entidades responsáveis por todas as regras de negócio da aplicação (classes Java convencionais, que usam e abusam de orientação a objetos); a camada de visualização, responsável pela interface com o usuário (em aplicações web, são os nossos arquivos JSP, HTML, CSS etc.); e a camada de controlador, que faz a ligação entre a interação do usuário na camada de visualização e as regras de negócio que vivem no modelo.

Ter essas camadas nos ajuda a separar melhor cada uma das responsabilidades e, por consequência, ter um código mais limpo, reusável e fácil de ser mantido.

2.5 Quando usar métodos privados?

Pensar com detalhes em cada método público, seu nome e assinatura é fundamental. Afinal, são eles que serão consumidos pelas classes clientes. Já métodos privados têm outro foco. A ideia deles é garantir a legibilidade do método público. É facilitar a vida do desenvolvedor na hora de ler código. Ao ler uma única linha, a que invoca o método privado, em vez das 30 linhas da sua implementação, você poupou algum tempo.

Veja um exemplo. O método `analisa()` é responsável por capturar arquivos `.java` em um diretório específico e depois, para cada arquivo, contar a quantidade de `if`s dentro dele:

```
class AnalisadorDeCodigo {
    public int analisa() {
        List<File> todosArquivos = arquivos.todosJava();

        int qtdIfs = 0;
        for(File arquivo : todosArquivos) {
            String codigo = IOUtils.readFile(arquivo);
```

```
            String[] tokens = codigo.split(" ");


            for(String token : tokens) {
                if(token.equals("if")) qtdIfs++;
            }

        }

        return qtdIfs;

    }
}
```

Veja que esse código é bastante coeso. Ele faz apenas uma única análise. Mas, mesmo assim, podemos dividi-lo em algumas partes:

I) captura de todos os arquivos;

II) loop em cada arquivo;

III) análise de cada arquivo.

Todas essas partes gastam apenas uma linha de código, com exceção da análise, que ocupa o corpo inteiro do `for`.

Ao olhar para esse código, um desenvolvedor novo do projeto precisará entender o que está acontecendo ali. É difícil dizer de primeira que ali é o código responsável por analisar cada código-fonte. Poderíamos extrair o código para uma outra classe, mas não é necessário: a classe `AnalisadorDeCódigo` já é simples o suficiente.

Podemos então extrair esse pedaço para um método privado e melhorar a legibilidade do método público. Precisamos fazer algumas mudanças, como colocar a variável `qtdDeIfs` como um atributo da classe, ou mesmo fazer o método privado devolver um inteiro para irmos acumulando.

A solução nesse momento não importa, contanto que você perceba que a legibilidade do método público ficou muito melhor. É fácil ver agora que estamos passeando por cada arquivo, e analisando um a um. Se quisermos

saber como essa análise funciona, aí sim mergulharemos na implementação do método `contaIfs()`.

```
class AnalisadorDeCodigo {
    private int qtdIfs = 0;

    public int analisa() {
        List<File> todosArquivos = arquivos.todosJava();

        for(File arquivo : todosArquivos) {
            contaIfs(arquivo);
        }

        return qtdIfs;
    }

    private void contaIfs(File arquivo) {
        // código do analisador aqui
    }
}
```

Métodos privados servem de apoio aos métodos públicos. Eles ajudam a aumentar a legibilidade de nossos códigos. Mas lembre-se: se seu método público crescer muito ou sofrer alterações constantes, talvez valha a pena extraí-lo para uma classe específica. Não use métodos privados quando o trecho de código representa uma outra responsabilidade da classe, mas sim para melhorar a legibilidade de algum algoritmo que é, por natureza, extenso ou complicado.

2.6 Falta de coesão em controllers

Um exemplo bem comum de falta de coesão são aqueles longos códigos escritos nos controladores de aplicações MVC. *Controllers* são aquelas classes que servem para conectar o mundo web, HTTP etc. com o mundo das regras de negócio, feitas em Java, banco de dados etc. É basicamente um conversor de um mundo para outro.

Mas, na prática, vemos muitas regras de negócio e código de infraestrutura misturados. Os problemas disso, você já conhece. Veja, por exemplo, o

trecho de código:

```
@Path("/notaFiscal/nova")
public void cadastraNotaFiscal(NotaFiscal nf) {
    if(nf.ehValida()) {

        // faz alguma regra de negocio qualquer
        if(nf.ehDeSaoPaulo()) {
            nf.duplicaImpostos();
        }

        if(nf.ultrapassaValorLimite()) {
            SMTP smtp = new SMTP();
            String template = leTemplateDoArquivo();
            smtp.enviaEmail(nf.getUsuario(), template);
        }

        // persiste no banco de dados
        String sql = "insert into NF (...) values (...)";
        PreparedStatement stmt = conexao.preparaSql(sql);
        stmt.execute();

        // envia nf pra um webservice qualquer
        SOAP ws = new SOAP();
        ws.setUrl("http://www.meuerp.com.br");
        ws.setPort(80);
        ws.send(nf);

        // exibe pagina de sucesso
        return view("sucesso.jsp");

    } else {
        // exibe pagina de erro
        return view("erro-de-valicacao.jsp");
    }

}
```

Observe que, em um único método, temos regras associadas a envio de e-mail, acesso a banco de dados e serviços web. O código de exemplo aqui

ainda é pequeno por questões de didática e espaço, mas imagine que gastamos muito mais do que 2 ou 3 linhas para acesso ao banco, serviço web, regras de negócio etc. É muita coisa para um único trecho de código. Diferente do exemplo anterior, um não tem relação com outro; são apenas processos de negócio diferentes que devem ser disparados de maneira sequencial.

Para resolver isso, não há segredo. Precisamos quebrar esse método pouco coeso, em outros métodos (ou mesmo classes) mais coesos. As regras de acesso a banco de dados podem estar encapsuladas em um `NotaFiscalDao`. O acesso ao serviço web poderia estar em uma outra classe, por exemplo, a `ERPInterno`. O mesmo para o envio de e-mail. Já ali, a sequência de regras de negócio podem estar bem divididas em um *Decorator*, ou algo que o valha.

Se fizermos isso, nosso código final será algo como:

```
@Path("/notaFiscal/nova")
public void cadastraNotaFiscal(NotaFiscal nf) {
    if(nf.ehValida()) {

        // regras de negocio, bem divididas
        // em classes
        RegrasDeCadastro regras =
            new DuplicaImpostoParaSaoPaulo(
            new EmailSeUltrapassaValorLimite());

        regras.aplica(nf);

        // persiste no banco de dados
        NotaFiscalDao dao = new NotaFiscalDao();
        dao.salva(nf);

        // envia nf pra um webservice qualquer
        WebService ws = new WebServiceDoERPInterno();
        ws.envia(nf);

        // exibe pagina de sucesso
        return view("sucesso.jsp");

    } else {
```

```
        // exibe pagina de erro
        return view("erro-de-valicacao.jsp");
    }

}
```

Veja que esse código apenas coordena um processo. Cada classe tem uma única responsabilidade e faz a sua parte. Agora, o código, do ponto de vista de coesão, já está muito melhor. As únicas regras de negócio que o controlador contém são regras de visualização. Ou seja, se "isso" acontecer, então exiba "aquilo".

Repare que esse exemplo é bastante comum em nosso dia a dia: classes que acessam banco de dados, tomam decisões de negócio e depois voltam ao banco de dados (ou a um serviço web). Sejam em aplicações MVC já um pouco mais modernas, ou mesmo naqueles códigos em ASP e PHP antigos, onde um arquivo tinha mil linhas de código, e fazia de tudo um pouco. Novamente, baixo reúso e dificuldade de manutenção imperam em códigos como esse.

Fuja de controladores "gordos" que fazem tudo. Separe bem cada uma das responsabilidades, sejam elas de negócio ou de infraestrutura.

Decorators e padrões de projeto

Não entrei em detalhes aqui na implementação do Decorator, pois ele tomaria espaço valioso desse livro. Se você ainda não conhece bem o padrão de projeto, sugiro ler mais sobre ele.

Conhecer vários deles o ajudará a ter um conjunto de boas soluções para seus problemas de coesão. Decorator, Chain of Responsibility, State, entre muitos outros, são padrões que nos ajudam a manter nossas classes coesas. Recomendo o livro do Eduardo Guerra, *Design Patterns com Java: Projeto orientado a objetos guiado por padrões*, sobre o assunto.

2.7 Inveja da outra classe

Controllers são sempre um bom exemplo onde programadores costumam escrever maus pedaços de código. Como vimos na seção, é muito fácil escrever controladores com regras de negócio. A boa prática então é fazer com que classes como essas apenas coordenem processos.

Mas, nessa tentativa, às vezes escrevemos classes "que invejam outras classes" (ou, no inglês, *feature envy*). Veja, por exemplo, o código abaixo, que mostra um método de um controlador, manipulando uma classe `Contrato` qualquer:

```
@Get("/contrato/fecha")
public void fecha(Contrato contrato) {
    contrato.setData("23/01/2015");
    contrato.fecha();
    List<Pagamento> pagamentos = contrato.geraPagamentos();

    if(contrato.isPessoaJuridica()) {
        contrato.marcaEmissaoDeNF();
    } else {
        contrato.marcaImpostoPF();
    }
}
```

Apesar desse código ter diversos problemas (que discutiremos à frente, como falta de encapsulamento), nesse momento, repare que ele só faz uso da classe `Contrato`. Isso é um indicativo de que talvez todo esse comportamento devesse estar dentro da própria classe de domínio.

Apesar do exemplo ter sido em um método de controlador, repare que esse mau cheiro pode ser encontrado em qualquer outro trecho de código do sistema. Não deixe suas classes invejarem o comportamento de outras.

2.8 SRP Single Responsibility Principle

O SRP, ou **Single Responsibility Principle**, é justamente o princípio que nos lembra de coesão. A descrição mais clara e formal do princípio diz que **a classe deve ter uma, e apenas uma, razão para mudar**. Como já disse

anteriormente, classes coesas tendem a ser menores e mais simples, menos suscetíveis a problemas, reúso mais fácil e a chance de propagarem problemas para outras classes é menor.

É realmente difícil enxergar a responsabilidade de uma classe. Talvez essa seja a maior dúvida na hora de se pensar em códigos coesos. É fácil entender que a classe deve ter apenas uma responsabilidade. O difícil é definir o que é uma responsabilidade, afinal é algo totalmente subjetivo. Por isso colocamos mais código do que deveríamos nela. **Dois comportamentos "pertencem" ao mesmo conceito/ responsabilidade se ambos mudam juntos**.

No exemplo do capítulo, deixei algumas ideias do que seriam classes coesas e como alcançá-las. Generalize a discussão que fiz. Como encontrar classes que não são coesas? Procure por classes que possuem muitos métodos diferentes; por classes que são modificadas com frequência; por classes que não param nunca de crescer.

Encontrou alguma classe não coesa, por meio dos padrões que mencionei anteriormente, ou mesmo porque você conhece bem o domínio? Então comece a pensar em dividir essas responsabilidades em classes menores. No exemplo dado, tínhamos regras parecidas e conseguimos criar uma abstração para representar. Mas, às vezes, não. Às vezes, o código possui responsabilidades totalmente distintas, e precisam apenas serem separadas. Se uma mesma classe possui código que acessa o banco de dados e também possui regras de negócio, apenas separaremos ambos os trechos de código e, por meio de composição, uniremos ambos os comportamentos.

Não ache que você conseguirá escrever classes coesas o tempo todo, e de primeira. Escrever código de qualidade é sempre incremental; você modela, observa seu modelo, aprende com ele e o melhora.

Preciso mesmo sempre ter classes pequenas?

Essa é uma ótima pergunta. Será que a implementação inicial, com os `if`s é sempre prejudicial? Os `if`s são tão ruins como falamos?

Lembre-se que, para modelar sistemas orientados a objetos, você precisa fazer trocas. A implementação final sugerida aqui no capítulo facilita a evolução e criação de novas regras de cálculo; é uma vantagem. A desvantagem é que, sem dúvida, ela é mais complexa do que o código procedural com `if`s.

Ou seja, leve isso em consideração. Se o seu problema for simples, talvez a solução com `if` não seja lá a pior do mundo. Encontrar os pontos onde sua modelagem precisa ser flexível e onde não precisa é um desafio. E dos bons.

2.9 Separação do modelo, infraestrutura, e a tal da arquitetura hexagonal

Se você trabalha com algum sistema legado, seja ele em VB, Delphi, JSF ou qualquer outra maluquice da JavaEE, sabe do que estou falando. É muito comum que o desenvolvedor escreva códigos totalmente acoplados à infraestrutura que está por baixo. Se o framework não permite que a classe tenha construtores, então ele não usa construtores. Se o framework pede para que a lógica de negócio fique acoplada ao banco de dados, ele acopla um ao outro. E assim por diante.

O problema disso? É que tudo fica mais difícil. Abrimos mão de classes coesas e flexíveis para ganhar os benefícios do framework. Escrevemos classes difíceis de serem testadas e mantidas pela simples questão da produtividade.

A sugestão é para que você separe ao máximo toda e qualquer infraestrutura (seja sem seu framework MVC, ou seu framework de persistência ou mesmo a biblioteca que você usa para criar serviços web) dos seus modelos e regras de negócio. Nem que, em último caso, você precise escrever uma camada de adaptação para sua infraestrutura, que é mais exigente do que deveria.

Lembre-se: não é o seu framework MVC que vai fazer seu software parar, e nem ele que será mantido no futuro. São suas regras de negócio. Essas classes é que precisam de atenção. A regra é simples: se a classe tem algum contato com infraestrutura, você não escreve regras de negócio alguma nelas; se a classe tem regras de negócio, ela não deve conhecer nenhuma infraestrutura.

Controllers são um bom exemplo da camada que fica no meio. Eles fazem a ponte entre a parte web e a parte de domínio. Eles contêm apenas transformações de um lado para o outro. Seus DAOs são outro bom exemplo. Eles devem apenas acessar o banco de dados, sem ter nenhuma regra de negócio.

Agora que você entendeu que precisamos separar infraestrutura de classes de modelo, você entende o que é a tal da arquitetura hexagonal (ou *ports and adapters*). Nesse tipo de arquitetura, separamos as portas (classes do domínio) de adaptadores (classes que fazem a ponte entre mundos diferentes, como web e domínio, banco de dados e domínio etc.).

Repare que, no fim, tudo é questão de separação. Se separarmos bem nosso código, ele será muito mais fácil de ser mantido e estendido. Se você depende de alguma infraestrutura (e provavelmente depende), lembre-se ao máximo de não a misturar com o resto do sistema.

2.10 Conclusão

Classes coesas são mais fáceis de serem mantidas, reutilizadas e tendem a ter menos bugs. Pense nisso.

Coesão é fundamental. Mas acoplamento também. Essa é uma balança interessante. Falaremos mais sobre o assunto no próximo capítulo.

Capítulo 3

Acoplamento e o tal do DIP

No capítulo anterior, discutimos sobre coesão. A ideia é que, se estivermos frente a uma classe com muitas responsabilidades, devemos dividir essas responsabilidades em muitas pequenas classes. Essa separação é importante do ponto de vista de manutenção, mas o software precisa juntar todos esses pequenos comportamentos e compor o comportamento maior, desejado pelo usuário. É nessa hora que complica o outro lado da balança: o acoplamento.

Acoplamento é um termo muito comum entre os desenvolvedores, em especial entre aqueles que programam usando linguagens OO. Até porque tem aquela grande frase, a máxima da orientação a objetos, que é **tenha classes que são muito coesas e pouco acopladas.** Mas a grande pergunta é: **por que o acoplamento é tão problemático?**

Veja o trecho de código a seguir:

```
public class GeradorDeNotaFiscal {
```

```
    private final EnviadorDeEmail email;
    private final NotaFiscalDao dao;

    public GeradorDeNotaFiscal(EnviadorDeEmail email,
        NotaFiscalDao dao) {
        this.email = email;
        this.dao = dao;
    }

    public NotaFiscal gera(Fatura fatura) {

        double valor = fatura.getValorMensal();

        NotaFiscal nf = new NotaFiscal(
            valor,
            impostoSimplesSobreO(valor)
        );

        email.enviaEmail(nf);
        dao.persiste(nf);

        return nf;
    }

    private double impostoSimplesSobreO(double valor) {
        return valor * 0.06;
    }
}
```

3.1 Qual o problema dela?

A classe `GeradorDeNotaFiscal` é acoplada ao `EnviadorDeEmail` e `NotaFiscalDao`. Pense agora o seguinte: hoje, esse código em particular manda e-mail e salva no banco de dados usando um DAO. Imagine que amanhã esse mesmo trecho de código também mandará informações para o SAP, disparará um SMS, consumirá um outro sistema da empresa etc. A classe `GeradorDeNotaFiscal` vai crescer, e passar a depender de muitas outras

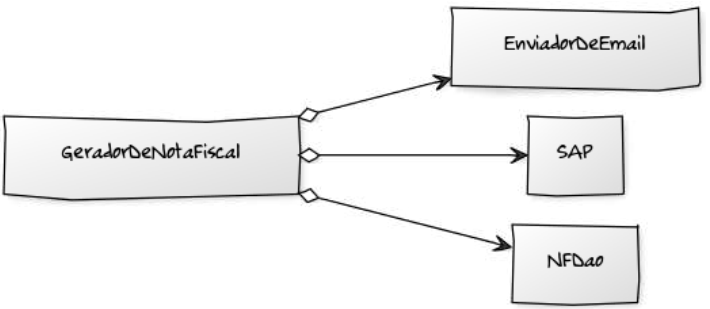
classes.

Qual o problema disso? O grande problema do acoplamento é que uma mudança em qualquer uma das classes pode impactar em mudanças na classe principal. Ou seja, se o `EnviadorDeEmail` parar de funcionar, o problema será propagado para o `GeradorDeNotaFiscal`. Se o `NFDao` parar de funcionar, o problema será propagado para o gerador. E assim por diante.

Podemos pensar não só em defeitos, mas em problemas de implementação. Se a interface da classe `SAP` mudar, essa mudança será propagada para o `GeradorDeNotaFiscal`. Portanto, o problema é: **a partir do momento em que uma classe possui muitas dependências, todas elas podem propagar problemas para a classe principal**.

O reúso dessas classes também fica cada vez mais difícil, afinal, se quisermos reutilizar uma determinada classe em outro lugar, precisaremos levar junto todas suas dependências. Lembre-se também que as dependências de uma classe podem ter suas próprias dependências, gerando uma grande árvore de classes que devem ser levadas junto.

É exatamente por tudo isso que o acoplamento é ruim. **A classe, quando possui muitas dependências, torna-se muito frágil, fácil de quebrar**.



Agora a próxima pergunta é: **será que conseguimos acabar com o acoplamento?** Ou seja, fazer com que as classes não dependam de nenhuma outra? É impossível. Nós sabemos que, na prática, quando estamos fazendo sistemas de médio/grande porte, as dependências existirão. O acoplamento

vai existir. Uma classe dependerá de outra que, por sua vez, dependerá de outra, e assim por diante.

Já que não é possível eliminar os acoplamentos, é necessário diferenciá-los. Afinal, será que todo acoplamento é problemático igual? Ou será que alguns são menos piores que outros? Porque, caso isso seja possível, modelaremos nossos sistemas fugindo dos "acoplamentos perigosos". Esse é o ponto chave deste capítulo.

Por mais estranho que pareça, é comum nos acoplarmos com classes e nem percebermos. Listas em Java, por exemplo. É comum que nossos códigos acoplem-se com a interface `List`. Ou mesmo com a classe `String`, muito utilizada no dia a dia. Ao usar qualquer uma das classes, seu código passa a estar acoplado a ele. Mas por que acoplar-se com `List` e `String` não é problemático, mas acoplar-se com `EnviadorDeEmail` ou com qualquer outra classe que contenha uma regra de negócio é?

Qual é a característica de `List` e qual é a característica de `String` que faz com que o acoplamento com ela seja menos dolorido do que com as outras classes? Encontrar essa característica é fundamental, pois aí bastará replicá-la; e, do mesmo jeito como não nos importamos ao acoplar com `List`, não nos importaremos em acoplar com outras classes dos nossos sistemas também.

O primeiro palpite é: *"Puxa, acoplar com List não é problema porque List é uma interface que o Java fez. Vem na linguagem Java*

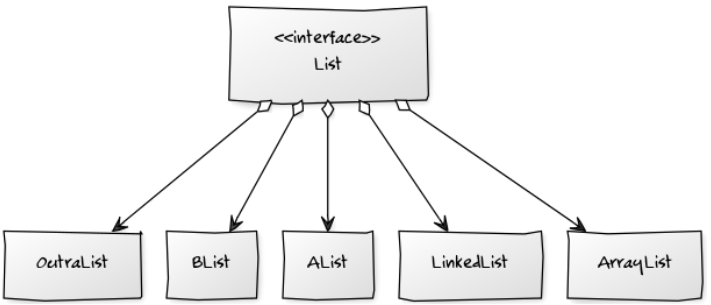
. A mesma coisa com a classe `String`, *"String vem com o Java*

. Mas não é bem essa a resposta.

3.2 Estabilidade de classes

A resposta, na verdade, é a interface `List` é **estável**. Ela muda muito pouco, ou quase nunca muda. E, como ela quase nunca muda, ela raramente propaga mudanças para a classe principal. Esse é o tipo de "acoplamento bom": a dependência é estável.

Mas por que a interface `List` é estável? Veja só quantas implementações dela existem: `ArrayList`, `LinkedList`, as várias implementações do Google dela etc. São várias. E quantas classes fazem uso dela? Provavelmente todos os sistemas Java que rodam hoje em nosso planeta.



Agora, imagine que você é desenvolvedor da linguagem Java: você teria coragem de mudar a interface `List`? É claro que não! Porque você sabe que essa mudança é difícil. Mudar a interface `List` implica em mudar a classe `ArrayList`, a classe `LinkedList`, em mudar o meu e o seu sistema. Essa "importância" faz dela estável. Por isso, ela tende a mudar muito pouco. Se ela tende a mudar muito pouco, quer dizer que a chance de ela propagar um erro, uma mudança, para a classe que a está usando é menor.

Ou seja, se uma determinada classe depende de `List`, isso não é um problema porque ela não muda. Se ela não muda, a classe principal não sofrerá impacto com a mudança dela. Este é o ponto: é acoplar-se a classes, interfaces, módulos, que sejam estáveis, que tendam a mudar muito pouco.

Repare que o que acabamos de discutir é um lado do acoplamento ao qual poucos olham. Acoplamento é geralmente olhar quais classes de que uma determinada classe depende. Contar isso é importante para detectarmos classes muito acopladas. O outro lado, que é olhar quantas classes dependem de uma classe, nos ajuda a dizer se a classe é ou não estável.

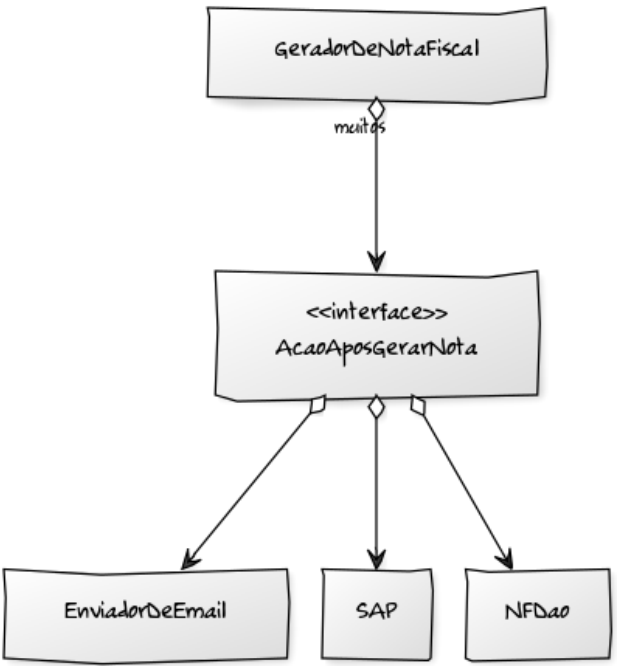
3.3 Buscando por classes estáveis

Como criar módulos estáveis, do mesmo jeito que a Oracle fez com `List`? Interfaces são um bom caminho pra isso. Afinal, interfaces são apenas contratos: elas não têm código que pode forçar uma mudança, e geralmente tem implementações dela, e isso faz com que o desenvolvedor pense duas vezes

antes de mudar o contrato (como mostrado anteriormente).

Veja, se emergirmos uma interface `AcaoAposGerarNotaFiscal` em nosso sistema, e fizermos com que `SAP`, `EnviadorDeEmail`, `EnviadorDeSMS`, `NFDao`, ou qualquer outra ação que deva ser executa, implemente essa interface, ela será estável por natureza.

Repare que a chance de ela mudar é baixa. Porque você, programador, vai ter medo de mexer nela. Se mexer nela, criando um método a mais, ou mudando uma assinatura de algum método, será necessário propagar essa mudança em todas as implementações. A interface também provavelmente será coesa, afinal, se ela tem 4, 5 implementações, é porque seu contrato é simples e bem definido. E interfaces coesas tendem a mudar menos.



É por isso que interfaces possuem um papel muito importante em sistemas orientados a objetos. A ideia de "programe voltado para interfaces" faz

todo sentido. Além de ganharmos flexibilidade afinal, podemos ter várias implementações daquela interface , a interface tende a ser estável. E se ela é estável, acoplar-se com ela é um problema menor.

Interfaces coesas

No capítulo anterior, discutimos sobre classes coesas. Mas por que não discutir também sobre interfaces coesas? Interfaces coesas são aquelas cujos comportamentos são simples e bem definidos. Suas implementações não precisam fazer "gambiarras" para se adaptarem.

Assim como nas classes, interfaces coesas tendem a mudar menos, e são mais reutilizáveis. Discutiremos mais sobre isso à frente.

Essa é a ideia para reduzir o problema do acoplamento. Não é deixar de acoplar. É começar a acoplar-se com módulos estáveis, que tendem a mudar menos. Interfaces são um bom exemplo disso.

Em código, a resposta para o problema do capítulo seria algo como o seguinte, onde criamos uma interface `AcaoAposGerarNota`, que representa a sequência de ações que devem ser executadas após a sua geração; a classe `GeradorDeNotaFiscal`, em vez de depender de cada ação específica, passa a depender de uma lista de ações. Repare que o gerador agora depende apenas da interface, que, por sua vez, é bastante estável. O problema está controlado:

```
public class GeradorDeNotaFiscal {

    private final List<AcaoAposGerarNota> acoes;

    public GeradorDeNotaFiscal(List<AcaoAposGeraNota> acoes) {
        this.acoes = acoes;
    }

    public NotaFiscal gera(Fatura fatura) {

        double valor = fatura.getValorMensal();

        NotaFiscal nf = new NotaFiscal(
```

```
            valor,
            impostoSimplesSobreO(valor)
        );

        for(AcaoAposGerarNota acao : acoes) {
            acoes.executa(nf);
        }

        return nf;
    }

    private double impostoSimplesSobreO(double valor) {
        return valor * 0.06;
    }
}

interface AcaoAposGerarNota {
    void executa(NotaFiscal nf);
}

class NFDao implements AcaoAposGerarNota {
    // implementacao
}

class QualquerOutraAcao implements AcaoAposGerarNota {
    // implementacao
}
```

Para os conhecedores de padrões de projeto, vejam que a solução é uma implementação do padrão de projeto *Observer*. *Observers* são uma ótima solução para o problema do acoplamento. Alguns padrões de projeto ajudam você a desacoplar seus projetos de classe, como o caso do *Observer*, *Visitor* e *Factory*. Estude-os.

Preciso então ter interfaces pra tudo?

É claro que não. Novamente você precisa pensar em cima do seu problema. Se você tem uma classe que sofre mudanças o tempo todo, pensar em melhorar as classes que se acoplam a ela pode ser uma boa ideia. Se ela não muda com tanta frequência, ou é uma classe da qual dependem poucas outras classes, talvez não seja necessário.

Programar voltado para interfaces também ajuda a flexibilizar. Veja que, em nosso código anterior, ficou fácil criar novas ações e plugá-las ao Gerador (falaremos mais sobre isso adiante). Mas, se você não precisar dessa flexibilidade, talvez não precise da interface também.

Lembre-se: estou mostrando soluções para problemas. Mas, antes de usar a solução, tenha certeza de que o problema existe.

3.4 DIP Dependency Inversion Principle

Agora que você já sabe o que é estabilidade, vamos falar do DIP, o Princípio da Inversão de Dependências. Neste capítulo, você percebeu que, se precisamos acoplar, que seja com classes estáveis. Podemos generalizar isso.

A ideia é: **sempre que uma classe for depender de outra, ela deve depender sempre de outro módulo mais estável do que ela mesma.** Se `A` depende de `B`, a ideia é que `B` seja mais estável que `A`. Mas `B` depende de `C`. Logo, a ideia é que `C` seja mais estável que `B`. Ou seja, suas classes devem sempre andar em direção à estabilidade, depender de módulos mais estáveis que ela própria.

Mas como conseguir isso? Lembre-se que abstrações tendem a ser estáveis, e implementações instáveis. Se você está programando alguma classe qualquer com regras de negócio, e precisa depender de outro módulo, idealmente esse outro módulo deve ser uma abstração. Tente ao máximo não depender de outras implementações (afinal, elas são instáveis).

Agora, se você está criando uma abstração e precisa depender de algum outro módulo, esse outro módulo também precisa ser uma abstração. Abstrações não devem depender de implementações! Abstrações devem ser es-

táveis.

De maneira mais elegante, o princípio diz:

- Módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstrações.
- Abstrações não devem depender de detalhes. Detalhes devem depender de abstrações.

Isso é o que nós chamamos de *Dependency Inversion Principle*, o Princípio de Inversão de Dependência. Não confunda isso com "injeção" de dependência. Injeção de dependência é a ideia de você ter os parâmetros no construtor, e alguém, geralmente um framework, automaticamente injetar essas dependências pra você. O nome é parecido. Aqui é o princípio da *inversão* de dependência. Você está invertendo a maneira de você depender das coisas. Passa a depender agora de abstrações.

Veja que nosso código segue o DIP. Afinal, a abstração `AcaoAposGerarNota` é uma abstração, estável, e não conhece detalhes de implementação. Já a classe `GeradorDeNotaFiscal` é uma implementação, o que faz dela um módulo mais instável, mas que só depende de abstrações.

Todas as classes devem ser estáveis?

Não! Se isso acontecesse, ou seja, todas as nossas classes fossem estáveis ao máximo, não conseguiríamos mexer em nada! Estaríamos engessados!

Devemos balancear entre módulos estáveis, em que não queremos mexer nunca, pois eles são importantes e muitas outras classes dependem deles, e módulos mais instáveis, que dependem dos estáveis, mas que vez ou outra precisam sofrer alterações. Essa é outra balança para você pensar: módulos estáveis e módulos instáveis.

3.5 Um outro exemplo de acoplamento

Discutimos até então um tipo de acoplamento que era mais difícil de ser resolvido. Aquele em que precisávamos lidar com interfaces e abstrações para conseguirmos desacoplar nosso código de maneira elegante e flexível. Mas nem todo acoplamento precisa de uma solução tão rebuscada.

Veja a classe `DespachadorDeNotasFiscais` a seguir. Ela é responsável por coordenar um processo complicado de despacho de notas fiscais. Ela calcula o imposto, descobre o tipo de entrega, despacha a entrega, e persiste a nota fiscal no banco de dados:

```
class DespachadorDeNotasFiscais {

    private NFDao dao;
    private CalculadorDeImposto impostos;
    private LeiDeEntrega lei;
    private Correios correios;

    public GerenciadorDeNotasFiscais(
        NFDao dao,
        CalculadorDeImposto impostos,
        LeiDeEntrega lei,
        Correios correios
    ) {
        this.dao = dao;
        this.impostos = impostos;
        this.lei = lei;
        this.correios = correios;
    }

    public void processa(NotaFiscal nf) {

        double imposto = impostos.para(nf);
        nf.setImposto(imposto);

        if(lei.deveEntregarUrgente(nf)) {
            correios.enviaPorSedex10(nf);
        } else {
            correios.enviaPorSedexComum(nf);
```

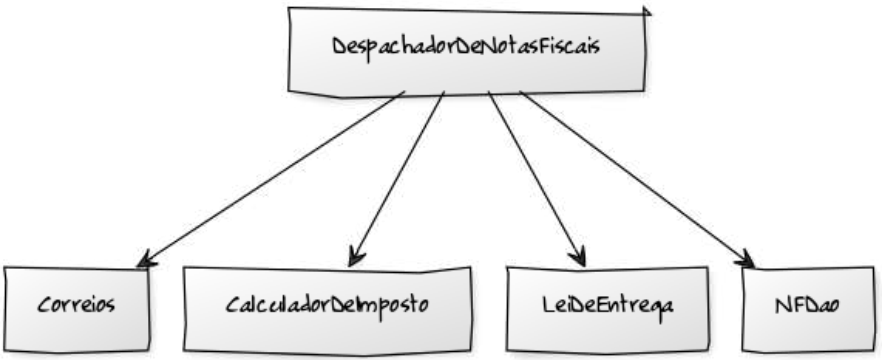
```
        }

        dao.persiste(nf);

    }
}
```

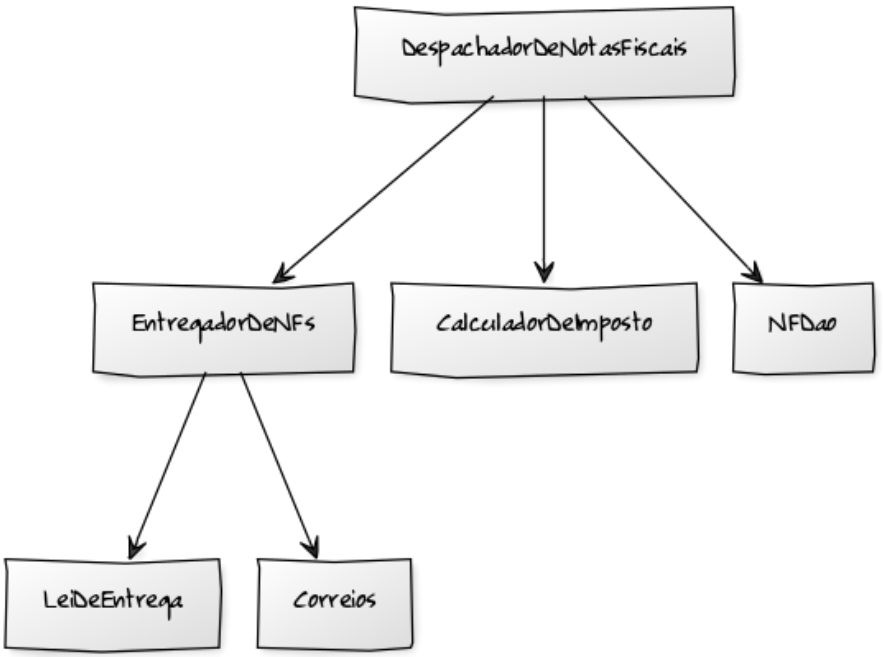
Para fazer tudo isso, ela depende de 4 outras classes. Mas, diferentemente do caso anterior, não há qualquer semelhança entre as dependências. Todas elas fazem coisas diferentes, dependendo inclusive de decisões tomadas ao longo do processo. Ou seja, aqui não conseguimos criar uma abstração única, estável, e resolver o problema.

Em casos como esse, a solução é pensar não só no acoplamento, mas também em divisão de responsabilidades. Veja o trecho que decide se a nota fiscal será enviada por SEDEX 10 ou por SEDEX Comum. Ela faz uso da `lei` para decidir qual método dos `correios` invocar. A variável `lei` é usada somente para isso:

```
if(lei.deveEntregarUrgente(nf)) {
    correios.enviaPorSedex10(nf);
} else {
    correios.enviaPorSedexComum(nf);
}
```

Por que não criar uma classe cuja responsabilidade é fazer essa tarefa? Uma nova classe, `EntregadorDeNFs`, por exemplo, que encapsularia essa regra de negócio. Dessa forma, a classe `DespachadorDeNotasFiscais` para de depender de `LeiDeEntrega` e de `Correios`, e passa a depender somente de `EntregadorDeNFs`. Veja as duas imagens, mostrando o antes e o depois da refatoração:





A classe principal agora ficou mais simples, dependendo de menos

classes, e com menos regras de negócio. Também é mais fácil de ser testada, afinal usaremos menos *mocks*.

Veja que você pode pensar também em "agrupar" dependências. Dessa forma, você diminui o acoplamento e aumenta a coesão da classe. Obviamente, não há regras para essa refatoração. Você pode agrupar 2, 3, 4 outras dependências; tudo depende do seu contexto. Procure encontrar trechos iguais aos do exemplo, onde dois ou mais dependências são usadas para algo específico, e somente para aquilo.

Nesses casos, observar atentamente os métodos privados da classe podem ajudar. Geralmente agrupamos naturalmente as dependências que trabalham juntas em métodos privados. Se você usa alguma IDE esperta, como o Eclipse, ele até acaba sugerindo a assinatura desses métodos privados, de maneira a deixar bem claro quais as dependências aquele método usará.

Lembre-se que às vezes uma classe está altamente acoplada porque ela não é coesa. Agrupar dependências é, no fim, aumentar a coesão. Perceba o quanto é difícil separar acoplamento de coesão.

3.6 Dependências lógicas

Até então, enxergar acoplamento entra classes era fácil. Basta ver se uma depende da outra, estruturalmente falando. Conseguimos ver isso no próprio código-fonte. Podemos fazer a lista de `imports` de uma classe, por exemplo, e dizer que todas aquelas classes são dependências da classe atual. Mas alguns desses acoplamentos não aparecem de forma clara em nosso código-fonte.

Por exemplo, em uma aplicação web MVC, muitas alterações no controller são propagadas para determinadas JSPs. Em frameworks como Ruby on Rails, por exemplo, em que o nome do método bate com o nome do HTML, sempre que você mexe no método `lista`, você também mexe no arquivo `lista.html.erb`. Isso também é acoplamento, só que não é claro do ponto de vista estrutural.

Esse tipo de acoplamento é o que chamamos de **acoplamento lógico**. Muitos deles ocorrem por definições arquiteturais, como foi o caso da aplicação web mencionada. Mas, quando esse tipo de acoplamento existe, por um motivo que não sabemos qual é, pode ser perigoso. Afinal, quando um muda,

o outro precisa mudar junto. Ou seja, o acoplamento lógico pode nos indicar um mau projeto de classes, ou mesmo código que não está bem encapsulado.

3.7 Conclusão

Neste capítulo, discuti o problema do acoplamento e a problemática propagação de mudanças que ele pode gerar. Chegamos à conclusão de que acoplar a classes estáveis, ou seja, classes que tendem a mudar pouco, é a solução para reduzir o problema do acoplamento.

E você achava que era impossível diminuir o acoplamento?

Capítulo 4

Classes abertas e o tal do OCP

Até então, discutimos bastante sobre acoplamento e coesão. Chegamos à conclusão de que classes não coesas devem ter suas responsabilidades divididas em pequenas classes, e que classes devem tentar ao máximo se acoplar com classes que são estáveis, ou seja, mudam pouco.

Com esses conceitos na cabeça, estamos prontos para começar a pensar em criar sistemas que evoluam mais facilmente. Esse é um ponto importante: nosso código deve estar sempre pronto para evoluir. E essas evoluções devem ser naturais. O desenvolvedor não deve sentir a necessidade de modificar muitos arquivos diferentes, ou mesmo procurar (usando o `CTRL+F`, por exemplo) os lugares que devem ser alterados.

Vamos a um exemplo de código. Veja:

```
public class CalculadoraDePrecos {
```

```
    public double calcula(Compra produto) {
        TabelaDePrecoPadrao tabela = new TabelaDePrecoPadrao();
        Frete correios = new Frete();

        double desconto =
            tabela.descontoPara(produto.getValor());
        double frete = correios.para(produto.getCidade());

        return produto.getValor() * (1-desconto) + frete;
    }
}

public class TabelaDePrecoPadrao {
    public double descontoPara(double valor) {
        if(valor>5000) return 0.03;
        if(valor>1000) return 0.05;
        return 0;
    }
}

public class Frete {
    public double para(String cidade) {
        if("SAO PAULO".equals(cidade.toUpperCase())) {
            return 15;
        }
        return 30;
    }
}
```

O código é bem simples. Ele basicamente pega um produto da loja e tenta descobrir seu preço. Ele primeiro pega o preço bruto do produto, e aí usa a tabela de preços padrão (`TabelaDePrecoPadrao`) para calcular o preço; pode-se ter um eventual desconto. Em seguida, o código descobre também o valor do frete. Por fim, ele faz a conta final: valor do produto, menos desconto, mais o frete.

O código é bastante coeso. Temos classes com responsabilidades bem definidas. A `TabelaDePrecoPadrao` calcula o desconto do produto, a classe `Frete` calcula o frete, e a classe `CalculadoraDePrecos` coordena o

processo e faz a conta final. O acoplamento também está razoavelmente controlado: a calculadora depende apenas de outras duas classes (número baixo). Ambas as classes utilizadas têm "cara" de que são modificadas com alguma frequência, mas ao menos o contrato delas parece bem simples e estável.

4.1 Qual o problema dela?

Mas agora imagine que o sistema é mais complicado que isso. Não existe apenas uma única regra de cálculo de desconto, mas várias; e também não existe apenas uma única regra de frete, existem várias. Uma maneira (infelizmente) comum de vermos código por aí é resolvendo isso por meio de `if`s. Ou seja, o código decide se é a regra A ou B que deve ser executada. O código a seguir exemplifica os `if`s para diferentes tabelas de preços:

```
public class CalculadoraDePrecos {

    public double calcula(Compra produto) {

        Frete correios = new Frete();

        double desconto;
        if (REGRA 1){
            TabelaDePrecoPadrao tabela =
                new TabelaDePrecoPadrao();
            desconto = tabela.descontoPara(produto.getValor());

        }
        if (REGRA 2){
            TabelaDePrecoDiferenciada tabela =
                new TabelaDePrecoDiferenciada();
            desconto = tabela.descontoPara(produto.getValor());
        }
        double frete = correios.para(produto.getCidade());
        return produto.getValor() * (1 - desconto) + frete;
    }
}
```

Se esse número de regras for razoavelmente grande, a ideia não é boa: esse código ficará complicadíssimo, cheio de `if`s, e difícil de ser mantido; testar fica cada vez mais difícil, afinal a quantidade de caminhos a serem testados é grande; a classe deixará de ser coesa, pois conterá muitas diferentes regras (e você já sabe os problemas que isso implica).

Uma segunda implementação comum é colocar os `if`s dentro das classes específicas. Por exemplo, a classe `Frete` passaria a ter as diferentes regras de negócio:

```
public class Frete {

    public double para(String cidade) {
        if(REGRA 1)  {
            if("SP".equals(cidade.toUpperCase())) {
                return 15;
            }
            return 30;
        }

        if(REGRA 2) { ... }
        if(REGRA 3) { ...}
        if(REGRA 4) { ...}
    }
}
```

Essa solução também tem problemas. A complexidade ainda continua a crescer. Já está melhor, claro, afinal todas as regras de frete estão na classe certa, mas ainda assim esse código pode ser bastante complexo. A interface dessa classe também pode ficar complicada. Afinal, ela precisará saber qual regra aplicar, e isso pode fazer com que o desenvolvedor comece a receber parâmetros ou mesmo ter uma grande quantidade de métodos sobrecarregados na classe.

Perceba: a discussão o tempo inteiro é sobre como balancear entre acoplamento e coesão. Buscar esse equilíbrio é fundamental!

4.2 OCP Princípio do Aberto-Fechado

Evoluir o código anterior é mais complicado do que parece. Escrever o `if` é fácil; o difícil é saber onde mais alterar. Precisamos fazer com que a criação de novas regras seja mais simples, e que essa mudança propague automaticamente por todo o sistema.

Um outro conceito que nos ajuda a ter classes coesas e que evoluam mais fácil é pensar sempre em escrever classes que são "abertas para extensão", mas "fechadas para modificação" (sim, esse é o famoso *Open Closed Principle*). A ideia é que suas classes sejam abertas para extensão. Ou seja, estender o comportamento delas deve ser fácil. Mas, ao mesmo tempo, elas devem ser fechadas para alteração. Ou seja, ela não deve ser modificada (ter seu código alterado) o tempo todo.

Voltemos ao exemplo da calculadora. Veja que, nesse momento, se quisermos mudar a maneira com que o cálculo de frete é feito, precisamos pôr as mãos nessa classe. Como possibilitar que a regra de frete seja alterada sem a necessidade de mexer nesse código? O primeiro passo é criarmos uma abstração para o problema, e fazer com que essas abstrações possam ser injetadas na classe que as usa. Se temos diferentes regras de desconto e de frete, basta criarmos interfaces que as representam:

```
public interface TabelaDePreco  {
    double descontoPara(double valor);
}

public class TabelaDePreco1 implements TabelaDePreco { }
public class TabelaDePreco2 implements TabelaDePreco { }
public class TabelaDePreco3 implements TabelaDePreco { }

public interface ServicoDeEntrega  {
    double para(String cidade);
}

public class Frete1 implements ServicoDeEntrega {}
public class Frete2 implements ServicoDeEntrega {}
public class Frete3 implements ServicoDeEntrega {}
```

Com abstrações em mãos, agora é fazer com que a calculadora faça uso

delas. Além disso, já que temos diferentes implementações, é necessário também que a troca entre elas seja fácil. Para isso, a solução é deixar de instanciar as implementações concretas dentro dessa classe, e passar a recebê-las pelo construtor.

Sempre que instanciamos classes diretamente dentro de outras classes, perdemos a oportunidade de trocar essa implementação em tempo de execução. Ou seja, se instanciamos `TabelaDePreco1` diretamente no código da classe principal, será sempre essa implementação concreta que será executada. E não queremos isso, queremos conseguir trocar a tabela de preço quando quisermos. Portanto, em vez de instanciarmos as classes de maneira fixa, vamos recebê-las por construtores.

Veja que essa simples mudança altera toda a maneira de se lidar com a classe. Com ela "aberta", ou seja, recebendo as dependências pelo construtor, podemos passar a implementação concreta que quisermos para ela. Se passarmos a implementação `TabelaDePreco1`, e invocarmos o método `calcula()`, o resultado será um; se passarmos a implementação `TabelaDePreco2` e invocarmos o mesmo método, o resultado será outro.

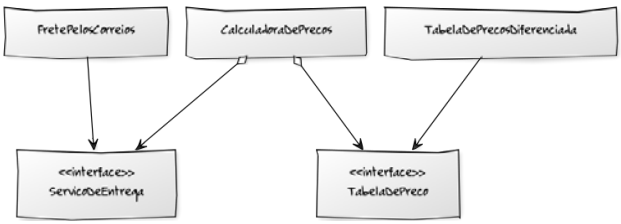
Ou seja, conseguimos mudar o comportamento final da classe `CalculadoraDePrecos` sem mudar o seu código. Como conseguimos isso? Justamente porque ela está aberta. É fácil mudar o seu comportamento interno, porque ela depende de abstrações e nos possibilita mudar essas dependências a qualquer momento.

Repare no código a seguir, que agora a classe recebe as dependências pelo construtor. Dessa forma, as classes clientes podem passar qualquer variável dessas dependências, mudando o comportamento final da classe principal:

```
// esse código desaparece
TabelaDePrecoPadrao tabela = new TabelaDePrecoPadrao();
Frete correios = new Frete();

public class CalculadoraDePrecos  {

    private TabelaDePreco tabela;
    private ServicoDeEntrega entrega;

    public CalculadoraDePrecos(
```

```
        TabelaDePreco tabela,
        ServicoDeEntrega entrega) {

        this.tabela = tabela;
        this.entrega = entrega;

    }

    // não instanciamos mais as dependências aqui,
    // apenas as usamos.
    public double calcula(Compra produto) {
        double desconto =
            tabela.descontoPara(produto.getValor());
        double frete = entrega.para(produto.getCidade());

        return produto.getValor() * (1-desconto) + frete;
    }
```

Em alto nível, veja só como estamos agora:



Só posso receber pelo construtor?

Existem diferentes maneiras de se injetar dependências em uma classe, como por exemplo, recebê-las pelo construtor, ou mesmo por *setters*.

Construtores geralmente são a melhor ideia. Afinal, eles obrigam os clientes dessa classe a passarem as dependências no momento da criação do objeto. Isso significa que nunca teremos um objeto instanciado sem as suas dependências. Repare que isso pode acontecer quando as injetamos por meio de *setters*. O programador precisa lembrar de usá-los. Já construtores não; o compilador fará o trabalho de avisá-lo sobre as dependências.

4.3 Classes extensíveis

Perceba que a classe anterior agora está aberta para extensão. Afinal, basta passarmos diferentes implementações de tabela e de frete para que ela execute de maneira distinta. Ao mesmo tempo, está fechada para modificação, afinal não há razões para mudarmos o código dessa classe. Essa classe agora segue o princípio do aberto-fechado.

O que discutimos aqui, de certa forma, mistura-se com a discussão do capítulo anterior sobre estabilidade e inversão de dependências. As interfaces (abstrações) `TabelaDePreco` e `ServicoDeEntrega` tendem a ser estáveis. A `CalculadoraDePrecos` é uma implementação mais instável e que só depende de abstrações estáveis. Pensar em abstrações nos ajuda a resolver o problema do acoplamento e, de quebra, ainda nos ajuda a ter códigos facilmente extensíveis.

Isso é programar orientado a objetos. É lidar com acoplamento, coesão, pensando em abstrações para nossos problemas. Quando se tem uma boa abstração, é fácil evoluir o sistema. Seu sistema deve evoluir por meio de novas implementações dessas abstrações, previamente pensadas, e não por meio de diversos `if`s espalhados por todo o código.

Por isso, pensar no projeto de classes é fundamental. A implementação

é, claro, importante, o código é o que dá vida à arquitetura pensada. Mas em um sistema OO, pensar no projeto de classes é o que garantirá a facilidade de manutenção.

If's nunca mais? Abstrações sempre?

Essa é também uma pergunta de um milhão de dólares. Até então vimos diversas maneiras de flexibilizar nossas classes, deixando-as abertas para extensão, criando abstrações etc. A pergunta é: devemos fazer isso o tempo todo?

Talvez não. Códigos flexíveis são importantes, têm um custo agregado: eles, com certeza, são mais complexos do que códigos não tão flexíveis. Olhar para um código, entender as abstrações que o permeiam e como as dependências interagem entre si pode não ser simples.

Muitas vezes um simples `if` resolve o problema. Portanto, seja parcimonioso. Flexibilize código que realmente precise disso, e seja simples em códigos que podem ser simples.

4.4 A testabilidade agradece!

A partir do momento em que a classe deixa clara todas as suas dependências, e possibilita a troca delas, criamos classes não só facilmente extensíveis, mas também altamente testáveis. Conseguimos escrever testes automatizados para a classe `CalculadoraDePrecos` sem nos preocuparmos com os comportamentos das dependências, pois conseguimos simulá-las por meio de *mock objects*. Mocks são objetos que dublam outros objetos, simplesmente para facilitar a escrita dos testes.

Mock Objects

Mock Objects é o nome que damos para objetos falsos, que simulam o comportamento de outros objetos. Eles são especialmente úteis durante a escrita de testes automatizados. Imagine que queremos testar nossa `CalculadoraDePrecos`. Para instanciar essa classe, precisamos também instanciar concretamente todas as suas dependências.

Qual o problema disso? Primeiro, se temos muitas classes concretas instanciadas, precisamos montar cenários concretos para todas elas, o que pode ser trabalhoso. Segundo, se o teste falhar, você não sabe quem causou a falha: foi a classe principal? Ou foi alguma das dependências?

É por isso que, ao fazer testes de unidade, geralmente simulamos (ou *mockamos*) as dependências. Dessa forma, conseguimos montar os cenários mais facilmente (afinal, o comportamento das dependências é simulado) e, se o teste falhar, a culpa é única e exclusivamente da classe que está sob teste.

O uso de mocks é bastante comum na indústria.

Veja o código a seguir, no qual testamos o comportamento do método `calcula()`. Sabemos que ele deve somar o valor do produto, subtrair o desconto dado e somar o valor do frete calculado. Mas, para testar esse comportamento sem depender da fórmula da tabela de descontos e do cálculo de entrega, podemos simular o comportamento das dependências, deixando o teste mais isolado e simples de ser escrito:

```
@Test
public void deveCalcularDescontoEFrete() {

    // criando os mocks
    TabelaDePreco simuladorDeTabela= mock(TabelaDePreco.class);
    ServicoDeEntrega simuladorDeEntrega =
        mock(ServicoDeEntrega.class);

    // cria a classe principal, passando os
    // mocks (dublês) como dependência
```

```
    CalculadoraDePrecos calculadora = new CalculadoraDePrecos(
        simuladorDeTabela,
        simuladorDeEntrega);

    Produto cd = new Produto("CD do Jorge e Mateus", 50.0);

    // simulando o comportamento das dependências
    // usando os mocks criados acima
    when(simuladorDeTabela.descontoPara(50)).thenReturn(5.0);
    when(simuladorDeEntrega.para("SP")).thenReturn(10.0);

    // invoca o comportamento que queremos testar
    double valor = calculadora.calcula(cd);

    // garante que o resultado da operação é 55 reais.
    assertEquals(55.0, valor, 0.0001);
}
```

Repare que o mesmo código de teste não seria possível se a classe não estivesse aberta. Como passaríamos os mocks para a classe? Nesse caso, precisaríamos testar a classe junto das suas dependências, o que pode ser complicado. Imagine que sua classe dependa de infraestrutura, como banco de dados ou serviços web.

É justamente por isso que a popular frase *"se está difícil de testar, é porque seu código pode ser melhorado*

faz total sentido. Um código fácil de ser testado é provavelmente um código bem projetado; já um código difícil de ser testado tem grandes chances de conter problemas de design.

É bastante simples de entender o motivo disso. O teste é simplesmente um pedaço de código que instancia uma classe, invoca um método com determinados parâmetros e verifica a sua saída. Se fazer somente isso está difícil é porque sua classe ou é altamente acoplada ou pouco coesa. E isso não é bom.

Em meu livro sobre TDD,*Test-Driven Development: Teste e Design no Mundo Real* discuto muito sobre a relação entre testabilidade e design, e é um ótimo complemento a esta leitura.

E quem injeta as dependências?

Uma pergunta que deve estar na sua cabeça nesse momento é: mas quem instancia as dependências quando "ligarmos" nosso software em produção? Sem dúvida alguma, alguém precisa fazer isso.

Temos geralmente duas saídas comuns: a primeira é usarmos fábricas (*factories*, do padrão GoF), ou mesmo algum framework de injeção de dependência. Discutiremos melhor isso nos capítulos à frente.

4.5 Um exemplo real

Usar tudo o que discutimos aqui não é fácil. Cair no código procedural é mais fácil do que parece, e é muitas vezes induzido pela dificuldade em lidar com a infraestrutura.

Veja, por exemplo, o código abaixo. Ele foi extraído de algum *commit* antigo da plataforma de ensino à distância da Caelum. Como toda plataforma de e-learning, ela tem sua representação para exercícios (a classe `Exercise`). Esses exercícios podem ser de tipos diferentes, e isso é feito por meio de herança (a classe `MultipleChoiceExercise` herda de `Exercise`, e assim por diante).

```
public class Exercise implements Identifiable {

    @Id
    @GeneratedValue
    protected Long id;

    @Valid
    private Text question = new Text(" ");

    @Valid
    @Embedded
    private HighlightedText highlightedQuestion =
        new HighlightedText(" ");

    @ManyToOne(fetch=FetchType.LAZY)
```

```
    private Section section;

    @OneToOne(fetch=FetchType.LAZY)
    private Discussion discussion;

    // e mais um monte de atributos

    // e mais um monte de comportamentos que os manipulam
}
```

O desafio é como fazer isso chegar à camada de visualização. Afinal, dependendo do tipo do exercício, a *jsp* a ser mostrada é diferente. Para isso, duas classes foram escritas. A classe `ShowAnswerHelper` contém funções auxiliares para a visualização:

```
@RequestScoped
public class ShowAnswerHelper {

    public String getUrlFor(Exercise exercise){
        if(exercise.getType().equals(Exercise.MULTIPLE_TYPE))
            return "/courses/"+exercise.getCourse().getCode()
            +"/sections/"+exercise.getSection().getNumber()
            +"/exercises/"+exercise.getNumber()
            +"/multipleChoiceAnswer";

        if(exercise.getType().equals(Exercise.CODE_TYPE)
        || exercise.getType().equals(Exercise.OPEN_TYPE))
            return "/courses/"+exercise.getCourse().getCode()
            +"/sections/"+exercise.getSection().getNumber()
            +"/exercises/"+exercise.getNumber()+"/openAnswer" ;

        return "/courses/"+exercise.getCourse().getCode()
        +"/sections/"+exercise.getSection().getNumber()
        +"/exercises/"+exercise.getNumber()
        +"/noAnswer" ;
    }

    public boolean isMultipleChoice(Exercise exercise){
        return exercise.getType()
```

```
                    .equals(Exercise.MULTIPLE_TYPE);
    }
    public boolean isOpenAnswer(Exercise exercise){
        return exercise.getType().equals(Exercise.OPEN_TYPE) ||
               exercise.getType().equals(Exercise.CODE_TYPE);
    }
    public boolean isNoAnswer(Exercise exercise){
        return exercise.getType().equals(Exercise.NO_ANSWER);
    }

}
```

E um arquivo *.tone*, que é similar a uma JSP. O funcionamento dele é parecido: as variáveis são declaradas no começo do arquivo, e depois código scriptlet em Java convencional dá o dinamismo necessário:

```
(@Exercise exercise)
(@inject ShowAnswerHelper answerUrlHelper)
(@inject AnswerStatusCheck statusCheck)

<%AnswerStatus status = statusCheck.analyze(exercise);%>

<%if(answerUrlHelper.isOpenAnswer(exercise)){%>
    <tone:uiShowOpenAnswerExercise status="@status"
    exercise="@exercise"
    url="@answerUrlHelper.getUrlFor(exercise)" />
<%}%>

<%if(answerUrlHelper.isMultipleChoice(exercise)){%>
    <tone:uiShowMultipleChoiseAnswerExercise
    status="@status"
    exercise="@exercise"
    url="@answerUrlHelper.getUrlFor(exercise)" />
<%}%>

<%if(answerUrlHelper.isNoAnswer(exercise)){%>
    <tone:uiShowNoAnswerExercise
    exercise="@exercise"
    url="@answerUrlHelper.getUrlFor(exercise)" />
```

```
<%}%>
```

Observe com atenção os dois últimos trechos de código. Repare que ambos contém sequências de `if`s, um para cada sub-tipo de `Exercise`.

Qual o problema disso? Repare que esse problema foi um dos mais discutidos nesse livro. A falta de abstração correta causa repetição de código. Note que toda vez que criarmos um novo tipo de exercício, teremos alterações nos dois arquivos mostrados. E repare que são os dois que apareceram aqui no livro. Como saber se não há mais pontos para mudar?

Esse código deixa bastante claro que precisamos deixar toda essa inteligência de visualização, de alguma forma, perto do exercício. E, ao criar novos tipos de exercícios, o desenvolvedor precisa pensar que criar um novo tipo implica em criar uma nova visualização também. Lembre-se que um bom projeto de classes é aquele que deixa claro qual o caminho a seguir.

Veja, por exemplo, o método `getUrlFor`. A primeira das mais simples refatorações é então colocar esse método em todos os exercícios. Porquê não transformar a classe `Exercise` em abstrata e deixar o método `getUrl()` abstrato?

```
public abstract Exercise {
    // atributos ...

    public abstract String getUrl();
}
```

Veja só que um simples código como esse resolve todo o imenso problema gerado pelo método `getUrlFor()`. Cada exercício tem o seu, o programador nunca vai esquecer de colocar a URL para novos exercícios, e não haverá mais a necessidade de colocar um `if` a mais naquele método. Ou seja, a classe não estará nunca fechada; ela sempre precisará ser aberta novamente para mais código. Isso quebra o OCP.

Mas será que vale a pena realmente colocar um conceito de visualização dentro de uma entidade de negócio? Isso não será uma quebra do princípio de responsabilidade única?

Sem dúvida, é. Mas às vezes não temos como fugir 100% disso. Uma boa solução é então tirar a implementação concreta de dentro dessas classes de

modelo e colocar em uma classe melhor isolada. Podemos criar, por exemplo, a interface `ExerciseViewDetails`, e fazer o `Exercise` devolvê-la:

```
interface ExerciseViewDetails {
    public String getUrl();
}

abstract class Exercise {
    public abstract ExerciseViewDetails viewDetails();
}
```

E dessa forma, cada exercício terá a sua própria implementação. O programador ainda assim continuará sem esquecer que isso é importante, afinal, sempre que criar um novo tipo, ele será obrigado a passar uma instância dessa nova interface:

```
class MultipleChoiceViewDetails implements ExerciseViewDetails {
    public String getUrl() {
        // implementacao aqui
    }
}

class MultipleChoiceExercise extends Exercise {
    public ExerciseViewDetails viewDetails() {
        return new MultipleChoiceViewDetails();
    }
}
```

Uma estratégia similar pode ser adotada para a JSP. O programador deve criar alguma abstração por cima daquilo e mantê-la perto do `Exercise` (justamente para evitar esses vários `if`s, um para cada subtipo).

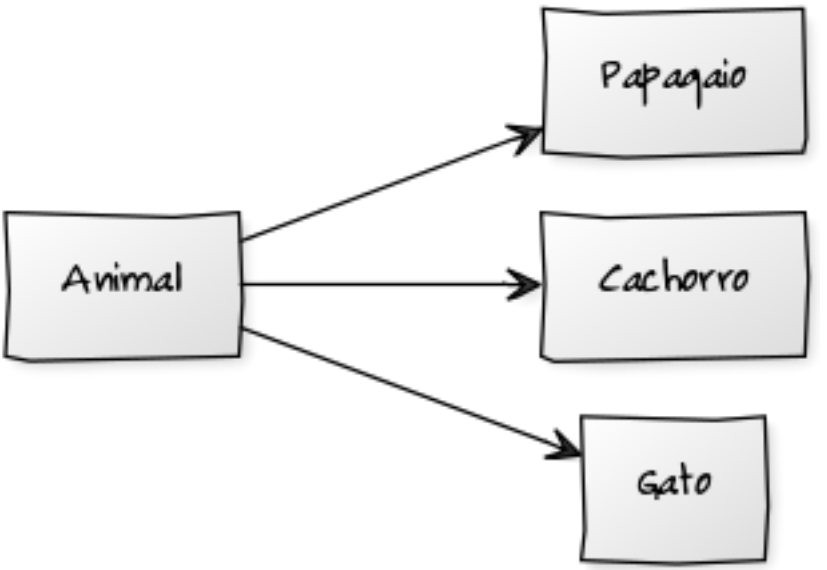
Obviamente, isso dependerá muito da sua infraestrutura. Nesse caso em particular, o programador precisa conhecer os detalhes do "tone", que é a infraestrutura de visualização, para descobrir como usar polimorfismo nela. Mas com certeza você encontrará uma maneira de contorná-la. Basta conhecer bem OO (e espero que esse livro esteja te ajudando nisso).

Veja então que, por simplesmente criarmos uma abstração, resolvemos o grande problema que é precisarmos propagar mudanças para diferentes pontos do código, onde muitos deles não são claros ou fáceis de encontrar.

4.6 Ensinando abstrações desde a base

Programar orientado a objetos é pensar em abstrações. Quando eu estou dando aula de Orientação a objetos básica, e o aluno está vendo pela primeira vez todos aqueles conceitos complicados de polimorfismo, herança, encapsulamento etc., uma brincadeira que eu sempre faço com eles é: no meio da aula eu falo "Gato, cachorro e pássaro". Eu espero que eles me respondam "animal" (a abstração mais adequada para o que falei). Eu viro e falo "ISS, INXPTO e outro-imposto-qualquer" e eu espero que a pessoa me responda "imposto". Eu faço o tempo inteiro o meu aluno pensar em abstração. Isso é programar orientado a objetos. É pensar primeiro na abstração, e depois, na implementação.

Essa é uma mudança de pensamento com quem programa procedural. Porque no mundo procedural, você está muito preocupado com a implementação. E é natural. No mundo OO, você tem que inverter: a sua preocupação maior tem que ser com a abstração, com o projeto de classes.



4.7 Conclusão

Neste capítulo, discutimos as vantagens de termos classes que são abertas para extensão e fechadas para modificação. Também discutimos como os conceitos agora começam a misturar: acoplamento, coesão e estabilidade.

Classes abertas são aquelas que deixam explícitas as suas dependências. Dessa maneira, podemos mudar as implementações concretas que são passadas para ela a qualquer momento, e isso faz com que o resultado final da sua execução mude de acordo com as classes que foram passadas para ela. Ou seja, conseguimos mudar o comportamento da classe sem mudar o seu código.

Lembre-se que sistemas OO evoluem por meio de novos códigos, e não de alterações em códigos já existentes. Programar OO é um desafio. Mas um desafio divertido.

Capítulo 5

O encapsulamento e a propagação de mudanças

Até então, discutimos acoplamento e coesão por vários pontos de vistas diferentes. Falta ainda um grande pilar, que é **encapsulamento**. Para facilitar a discussão, como sempre, vamos a um exemplo de código:

```
public class ProcessadorDeBoletos {

    public void processa(List<Boleto> boletos, Fatura fatura) {
        double total = 0;

        for(Boleto boleto : boletos) {

            Pagamento pagamento = new Pagamento(
                boleto.getValor(),
```

```
                MeioDePagamento.BOLETO);

            fatura.getPagamentos().add(pagamento);

            total += fatura.getValor();
        }

        if(total >= fatura.getValor()) {
            fatura.setPago(true);
        }
    }
}
```

A implementação é simples. O método `processa` deixa bem claro que, dadas uma lista de boletos e uma fatura, ele passeia por cada boleto, gera um pagamento e associa esse pagamento à fatura. No fim, ele faz uma verificação: se o valor da fatura for menor do que o total pago, quer dizer que a fatura já foi paga por completo.

Vamos analisar esse código, olhando para os aspectos que já conhecemos. Essa classe parece coesa. Ela é responsável por uma única regra de negócio, que não se encaixa bem em nenhuma outra entidade do sistema. Do ponto de vista de acoplamento, ela está acoplada apenas a entidades, e à interface `List`.

5.1 Qual o problema dela?

O problema desse código é a falta de encapsulamento. Encapsulamento é o nome que damos à ideia de a classe esconder os detalhes de implementação, ou seja, **como** o método faz o trabalho dele.

Mas por que esconder? Se a implementação está bem escondida dentro da classe, isso nos traz dois ganhos. O primeiro é a facilidade para alterar a implementação. Podemos, por exemplo, pegar todo o código dentro do método, jogar fora, e fazer de novo. O resto do sistema nem perceberá a mudança.

O segundo, e bem mais delicado, é que, se o código não está bem encapsulado, isso implica em termos a regra de negócio espalhada por lugares diferentes. Isso quer dizer que, sempre que a regra de negócio mudar, essa mu-

dança deverá ser propagada em muitos lugares diferentes. O grande problema é: como localizar esses lugares?

Um grande problema de sistemas legados

Sistemas legados são problemáticos por muitos motivos: código macarrônico, algoritmos complexos e pouco performáticos, muitos bugs etc. Aposto que todos nós temos algumas experiências interessantes com códigos legados.

Mas um dos piores problemas deles é não saber em quais classes mexer, dada uma funcionalidade que deve ser modificada. Programar usando o `CTRL+F`, buscando por lugares onde as mudanças devem ser feitas, é problemático. O desenvolvedor não vai encontrar todos os lugares e, quando colocar o software em produção, ele não funcionará corretamente. Aí o desenvolvedor voltará ao código e continuará buscando pelos pontos que esqueceu de alterar.

Código macarrônico é fácil de ser corrigido. Você passa um dia entendendo o código monstruoso, faz uma nova implementação mais concisa e clara, e descarta a antiga. Agora, não saber onde mexer e precisar procurar código dificultará de verdade a evolução do seu sistema.

Repare que, nesse código em particular, temos uma regra de negócio que está no lugar errado. O código responsável por marcar a fatura como paga ou não está no `ProcessadorDeBoletos`:

```
total += boleto.getValor();

}
if(fatura.getValor()<= total)  {
   fatura.setPago(true);
}
```

Essa é uma regra que não deveria estar no `ProcessadorDeBoletos`. Essa é uma regra da fatura, portanto, deveria estar escondida na classe `Fatura`. Novamente, qual é o problema dessa regra estar aí?

O problema é que amanhã, se aparecer o `ProcessadorDeCartaoDeCredito`, o desenvolvedor será obrigado a repetir esse código. E se tivermos 3 ou 4 lugares diferentes, que fazem uso da `Fatura`? No momento em que mudarmos essa regra a regra de marcar uma fatura como paga precisaremos sair buscando no código os lugares que têm essa regra. Tudo isso por quê? Porque essa regra não está escondida. A `Fatura` é que deve ser a responsável por se marcar como paga. Ela sabe o momento de estar paga e o momento de não estar paga. Até porque, repare que ela tem a lista de pagamentos. Essa regra poderia estar lá dentro, sem qualquer problema do ponto de vista de implementação.

Esse é um problema de encapsulamento. Uma classe (ou método) bem encapsulada é aquela que esconde bem a maneira como faz as coisas. A classe deve apenas deixar claro o que ela faz, quais são as operações que ela provê para as classes que farão uso dela.

5.2 Intimidade inapropriada

```
NotaFiscal nf = new NotaFiscal();
double valor;
if (nf.getValorSemImposto() > 10000)  {
    valor = 0.06 * nf.getValor();
}
else {
    valor = 0.12 * nf.getValor();
}
return valor;
```

Perceba que é um problema bastante análogo ao anterior. Temos aqui uma `NotaFiscal`, e um `if` cuja ideia é: "Olha, se o valor da nota sem imposto for maior que 10 mil, eu vou calcular o valor da nota de um jeito. Caso contrário, eu vou calcular de outro jeito".

Esse é outro exemplo de código que está mal encapsulado. A classe que contém esse código imagine que é uma outra classe qualquer, com esse trecho de código sabe demais sobre como funciona uma nota fiscal. E quem que deve saber como que funciona uma nota fiscal? A própria classe `NotaFiscal`.

Chamamos isso de *intimidade inapropriada*, os códigos como o anterior,

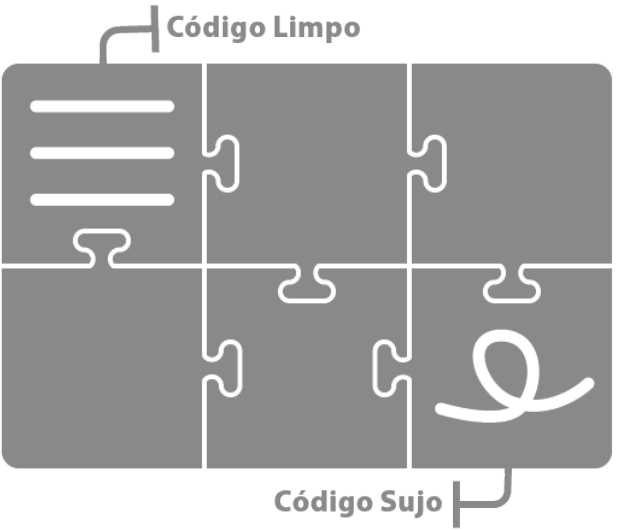
que entendem mais do que deveriam sobre o comportamento de uma outra classe. A solução para isso é a mesma: encapsular a fórmula de cálculo de valor de imposto dentro da classe `NotaFiscal`:

```
NotaFiscal nf = new NotaFiscal();
double valor = nf.calculaValorImposto();
```

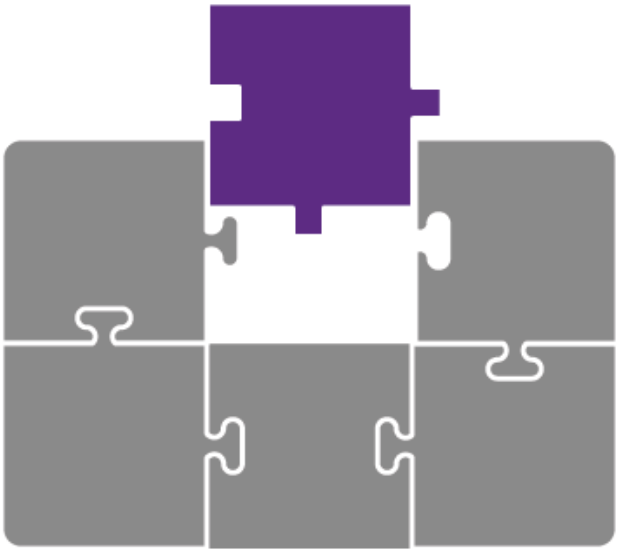
Agora a regra de calcular o imposto está escondida. Isso nos dá a possibilidade de alterar essa regra sem a necessidade de alterar qualquer outro ponto do sistema.

5.3 Um sistema OO é um quebra-cabeças

Uma analogia que me agrada bastante é pensar em um sistema como um grande quebra-cabeças. Cada peça é uma classe, e elas se encaixam umas as outras para formar algo maior. Os encaixes das peças devem ser bem pensados, afinal, mudá-los não é fácil. Imagine, se você mudar o formato de uma peça, você precisará propagar essa mudança para muitas peças ao redor. Agora, se uma das peças estiver com o seu desenho interno riscado, por exemplo, basta trocar aquela única peça. O resto do quebra-cabeças não será afetado.



Agora imagine que o formato da peça é a interface que essa classe provê às outras classes; o desenho é a implementação. Se a interface estiver clara, coesa e bem definida, você poderá trocar a implementação daquela classe sem afetar o resto do sistema.



É isso que fará a diferença em um sistema grande: classes mal implementadas são infelizmente comuns, mas são um problema menor quando o sistema está bem modelado. Pense sempre no formato das suas peças. Isso é programar OO: é pensar no formato das peças; às vezes, pensar até mais do que no próprio desenho.

5.4 Tell, Don't Ask

Um conhecido princípio de Orientação a Objetos é o *Tell, Don't Ask*, ou seja, "Diga, não pergunte". Mas, como assim, diga e não pergunte? Se relembrarmos o código anterior, perceba que a primeira coisa que fazemos para o objeto é uma pergunta (ou seja, um `if`) e, de acordo com a resposta, damos uma ordem para esse objeto: ou calcula o valor de um jeito ou calcula de outro.

```
NotaFiscal nf = new NotaFiscal();
double valor;
if (nf.getValorSemImposto() > 10000)  {
    valor = 0.06 * nf.getValor();
```

```
}
else {
    valor = 0.12 * nf.getValor();
}
```

Quando temos códigos que perguntam uma coisa para um objeto, para então tomar uma decisão, é um código que não está seguindo o *Tell, Don't Ask*. A ideia é que devemos sempre dizer ao objeto o que ele tem que fazer, e não primeiro perguntar algo a ele, para depois decidir. O código que refatoramos faz isso direito. Perceba que estamos dando uma ordem ao objeto: calcule o valor do imposto. Lá dentro, obviamente, a implementação será o `if` anterior, não há como fugir disso. Mas ele está **encapsulado** no objeto:

```
NotaFiscal nf = new NotaFiscal();
double valor = nf.calculaValorImposto();
```

Códigos como esse, que perguntam para depois tomar uma decisão, tendem a ser procedurais. Em linguagens como C, por exemplo, não há muito como fugir disso. Isso é programação procedural. No mundo OO, devemos o tempo todo dar ordens aos objetos. A partir do momento em que perguntamos primeiro para tomar uma decisão, provavelmente estamos furando o encapsulamento.

5.5 Procurando por encapsulamentos problemáticos

Perceber se um código está bem encapsulado ou não, não é tão difícil. Olhe, por exemplo, para o código da nota fiscal refatorado. Agora se pergunte:

`O que esse método faz?` Provavelmente sua resposta será: eu sei o que o método faz pelo nome dele, calcula o valor do imposto. É um nome bem semântico, deixa claro que ele faz. Se você conseguiu responder essa pergunta, está no bom caminho.

A próxima pergunta é: `Como ele faz isso?` Sua resposta provavelmente é: se eu olhar só para esse código, não dá para responder. Não dá para dizer qual é a regra que ele está usando por debaixo dos panos. Não sabemos

a implementação do `calculaValorImposto()`. Isso, na verdade, é uma coisa boa. Isso é encapsulamento.

Um exemplo bastante comum de código bem encapsulado e isso as pessoas acertam na maioria das vezes nos códigos OO são os DAOs. O DAO é aquela classe onde escondemos todo o código de acesso aos dados; na maioria das vezes, persistido em um banco de dados. O código a seguir mostra um exemplo de uso de um DAO convencional:

```
NotaFiscalDao dao = new NotaFiscalDao();
List<NotaFiscal> notasFiscais = dao.pegaTodos();
```

O que o método `pegaTodos` faz? Pega todas as notas fiscais. Como ele faz? Não sabemos! Não sabemos se vem de um banco de dados, se vem um serviço web, se ele está lendo um arquivo texto. Tudo isso está escondido dentro da implementação do método. Encapsulado.

Uma linha de raciocínio bastante interessante na hora de se programar OO é não pensar só na implementação daquela classe, mas também nas classes clientes, que a consumirão. Novamente, é isso que geralmente faz um sistema difícil de se manter. Sistemas difíceis de se manter são aqueles que não pensam na propagação de mudança. É um sistema em que você, para fazer uma mudança, tem que mudar em 10 pontos diferentes. Códigos bem encapsulados geralmente resolvem esse tipo de problema, porque você muda em um lugar e a mudança se propaga.

O teste é o primeiro cliente

No meu livro de TDD, comento que o teste de unidade é um excelente primeiro cliente de uma classe. O teste faz uso intenso da classe de produção, e nele você consegue ver os problemas de coesão, acoplamento e encapsulamento.

Não entrarei em mais detalhes sobre o assunto aqui, pois já escrevi um livro sobre isso. Mas lembre-se de sempre experimentar a classe que escreveu. No caso mais simples, escreva um simples método `main` que faz uso do método que você acabou de escrever. Aí, olhe para essa `main` e reflita.

5.6 A famosa Lei de Demeter

Encapsular pode ser mais difícil do que parece. Veja o trecho de código a seguir:

```
public void algumMetodo()  {
    Fatura fatura = pegaFaturaDeAlgumLugar();
    fatura.getCliente().marcaComoInadimplente();
}
```

Veja que, para marcarmos o cliente como inadimplente, primeiro foi necessário pegá-lo na fatura. Para tal, escrevemos `fatura.getCliente().marcaComoInadimplente();`. Generalize esse trecho de código. É bastante comum encontrarmos códigos que fazem `A.getB().getC().getD().metodoQualquer()`, ou seja, uma cadeia de invocações. Acredite ou não, um código desse pode estar quebrando o encapsulamento da classe.

Imagine só que, por algum motivo, a classe `Cliente` sofra uma mudança. E sofra o pior tipo de mudança possível: a sua interface pública perdeu o método `marcaComoInadimplente()`. Quais serão os pontos do sistema que deixarão de compilar por causa disso? O código provavelmente quebrará em todo lugar que usa a classe `Cliente` e em todo lugar que usa uma `Fatura`, e que usa um `Cliente` ao mesmo tempo. Ou seja, que usa a classe `Cliente` de maneira indireta, porque a classe `Fatura` permite isso.

Se você parar para pensar, esse código depende de `Fatura` de maneira direta, e de `Cliente` de maneira indireta, já que ele invoca `getCliente()`. É um tipo de acoplamento difícil de ser notado.

Esse é o problema de invocações em cadeia. Se temos `a.getB().getC().getD()`, se B mudar, ou se C mudar, ou se D mudar, esse código quebrará.

Como resolver o problema? Uma possível solução é encapsular melhor a classe `Fatura`. Os consumidores de `Fatura` não precisam saber o tempo todo que ela possui um `Cliente` dentro. Veja:

```
public void algumMetodo() {
    Fatura fatura = pegaFaturaDeAlgumLugar();
```

```
    fatura.marcaComoInadimplente();
}
```

O método `marcaComoInadimplente` tenta agora encapsular o processo de marcar o cliente como inadimplente. Dentro lá da fatura, o código obviamente fará `cliente.marcaComoInadimplente()`.

E se a classe `Cliente` mudar, o que acontecerá? Se ela mudar, a classe `Fatura` vai parar de funcionar. Mas agora mexeremos em um único lugar. Lembre-se que a ideia é sempre diminuir pontos de mudança. É melhor ter que mexer nas classes `Cliente` e `Fatura`, do que mexer em `Cliente`, `Fatura`, em todo mundo que mexe em `Cliente`, e em todo mundo que mexe em `Cliente` de maneira indireta através da `Fatura`. Diminua ao máximo pontos de mudança.

É isso que diz a *Lei de Demeter*. Ela sugere que evitemos o uso de invocações em cadeia, como a discutida aqui. O que ganhamos com isso? Encapsulamento. E o que eu ganho com encapsulamento? Diminuo propagação de mudanças.

Nunca devo fugir da Lei De Demeter?

Claro que, como qualquer outra regra, você não deve levá-la 100% à risca. Encadear *getters* para, por exemplo, exibir um dado na camada de visualização, não é tão problemático. Imagine uma JSP que deve exibir o endereço do cliente:

```
String rua = fatura.getCliente().getEndereco().getRua();
// imprime a variável rua
```

Ou seja, tenha essa lei na cabeça, e use-a (ou quebre-a) quando fizer sentido.

5.7 Getters e setters pra tudo, não!

Uma coisa interessante de toda primeira aula de Java é que o professor mostra o que é uma classe, discute que classes têm atributos e métodos, e modela

junto com o aluno um primeiro exemplo. Imagine por exemplo uma classe `ContaCorrente`, que deve armazenar informações como nome do cliente, número da agência, número da conta e saldo. Ele cria atributos para cada uma dessas informações e, logo na sequência, *getters* e *setters* para todos eles.

Por que criamos *getters* e *setters* pra todos atributos? Isso é saudável? *Getters* e *setters* podem ser bastante perigosos se criados sem parcimônia. A partir do momento em que o desenvolvedor dá um *setter* para um determinado atributo da classe, ele está dando a oportunidade de qualquer classe cliente modificar aquele valor de qualquer jeito. Imagine o método `setSaldo()`: qualquer classe cliente poderia definir um novo valor de saldo para um cliente. Imagino que em um banco, a quantidade de regras para se mexer no saldo de um cliente tendem ao infinito. Um `setSaldo` só iria facilitar a quebra do encapsulamento.

Imagine classes clientes fazendo coisas do tipo:

```
double saldoAtual = conta.getSaldo();
if(HOJE É NATAL) {
    novoSaldo = saldoAtual + 100.0;
}
conta.setSaldo(novoSaldo);
```

Apesar de ser uma regra interessante (afinal, quem não quer ganhar R$100,00 no Natal?), esse código tem os mesmos problemas discutidos até então. Falta de encapsulamento.

Portanto, antes de criar *setters*, pense se eles não gerarão problemas de encapsulamento no futuro. É preferível você fornecer comportamentos que alterem o valor do atributo. No exemplo da conta, poderíamos ter métodos como `saca()` ou `deposita()`, muito mais interessantes do que um *setter*.

O *getter* é menos prejudicial, afinal ele apenas devolve a informação para o usuário. E precisamos deles, afinal nossos sistemas precisam exibir os dados, ou mesmo recuperá-los para fazer algum tipo de processamento. Mas alguns deles podem ser perigosos também. Veja o trecho de código a seguir:

```
fatura.getPagamentos().add(pagamento);
```

Repare que o método `getPagamentos()` dá a liberdade de as classes clientes adicionarem pagamentos à lista. Provavelmente isso não é algo dese-

jável. Afinal, adicionar um pagamento em uma fatura pode ter regras associadas.

Uma boa ideia é fazer que seus *getters* sempre devolvam cópias dos objetos originais, ou mesmo bloqueiem alterações de alguma forma. Para esse problema de listas em particular, podemos abusar da API da linguagem Java, e devolver uma lista não modificável:

```
public List<Pagamento> getPagamentos()  {
    return Collections.unmodifiablelist(pagamentos);
}
```

Agora, se um cliente tentar inserir um pagamento diretamente na lista, ele receberá uma exceção. A única maneira de adicionar um pagamento deve ser por meio de um comportamento, bem encapsulado, na classe `Fatura`.

Novamente, lembre-se de não criar *getters* e *setters*: sem pensar. Eles precisam de um motivo para existir. Lembre-se que até esses simples métodos, que você aprendeu a criar na sua primeira aula de Java, podem tornar-se inimigos mais para frente.

5.8 Corrigindo o código inicial

É hora de encapsular o código inicial. Sabemos que o problema do encapsulamento está aqui:

```
if(total >= fatura.getValor()) {
    fatura.setPago(true);
}
```

A fatura não pode ser marcada como paga por esse código `ProcessadorDeBoletos`. Essa regra de negócios precisar estar encapsulada na classe `Fatura`. Ou seja, aqui dentro, eu tenho que achar um bom lugar pra colocar isso.

A primeira coisa que devemos fazer é remover o método `setPago()`. A regra para marcar uma fatura como paga ou não deve pertencer somente à `Fatura`, e qualquer classe não pode simplesmente marcar a fatura como paga.

Vamos adicionar um método `adicionaPagamento()` à classe `Fatura`, e esse método será responsável por adicionar pagamentos, e automaticamente verificar se a fatura deve ser marcada como paga. Veja o código:

```
public void adicionaPagamento(Pagamento pagamento)  {
    this.pagamentos.add(pagamento);

    if(valorTotalDosPagamentos()>this.valor)  {
        this.pago = true;
    }

}

private double valorTotalDosPagamentos()  {
    double total = 0;

    for(Pagamento p : pagamentos)  {
    total += p.getValor();
    }
    return total;
}
```

Pode não ser a melhor implementação possível, afinal temos um loop, e dentro dele, outro loop. Mas lembre-se sobre o que discutimos: implementação com código feio, difícil de ser lido, ou com um algoritmo mais complicado que deveria ser, é um problema fácil de resolver. Se tudo está bem encapsulado, basta você trocar a implementação por uma melhor.

Veja agora nosso novo `ProcessadorDeBoletos`. Ao olharmos a invocação do método `adicionaPagamento()`, sabemos o que ele faz, mas não como ele faz seu trabalho. Encapsulado!

```
public class ProcessadorDeBoletos {

    public void processa(List<Boleto> boletos, Fatura fatura) {

        for(Boleto boleto : boletos) {
                    Pagamento pagamento = new Pagamento(
                     boleto.getValor(),
```

```
                    MeioDePagamento.BOLETO
                  );

        fatura.adicionaPagamento(pagamento);
        }
    }
}
```

Encapsulamento é esconder como a classe implementa as suas tarefas. Como que eu sei que as minhas classes e métodos estão encapsulados? Basta olhar para ela e tentar responder as duas perguntas: O quê? E como?

O "o quê?" você tem que ser capaz de responder, porque o nome do método tem que lhe dizer isso. O "como?" você não tem que conseguir responder. Se conseguir responder, não está encapsulado. Resolva o problema de encapsulamento, e isso vai lhe garantir depois umas boas horas de sono, porque vai lhe dar menos trabalho para fazer uma mudança do seu usuário final.

5.9 Modelos anêmicos

Evite também os chamados modelos anêmicos, muito comuns no passado negro da JavaEE. OO é modelar o mundo real por meio de classes que contêm dados e comportamentos. Mas é impressionante a quantidade de aplicações que fogem disso. Muitos sistemas têm classes que, ou têm atributos, ou têm métodos. Nunca os dois juntos. Ou seja, temos código procedural em linguagem OO novamente.

A desculpa na época era porque a Sun divulgava isso como boas práticas de desenvolvimento. Naquele tempo, elas até faziam algum sentido quando você precisava criar sistemas distribuídos, e os EJBs ainda eram péssimos. Desenvolvedores resolveram adotar isso para todo e qualquer tipo de aplicação, sem pensar muito a respeito. A consequência? Sistemas anêmicos em todos os lugares.

O código a seguir exemplifica o que é um modelo anêmico. Veja que ele é exatamente o oposto de tudo que discutimos aqui. No exemplo, a classe `Fatura` contém apenas atributos, e a classe `FaturaBO` (que pode ter muitos

nomes por aí, como `FaturaDelegate`, `FaturaBLL` etc.) contém apenas regras de negócio. É, no fundo, um grande código procedural, onde temos estruturas que guardam dados e funções que manipulam essas estruturas:

```
class Fatura {
    private String cliente;
    private double valor;
    private List<Item> itens;

    // getters e setters

    // nenhum método de negócio
}

class FaturaBLL {

    public void finaliza(Fatura f) {
        // regra de finalização
    }

    public void calculaImposto(Fatura f) {
        // regra de impostos
    }

    // outros comportamentos aqui
}
```

Sem dúvida, é muito mais fácil escrever código procedural, afinal pensar em um projeto de classe é desafiador. O problema é que teremos, muito provavelmente, todos os problemas que discutimos no começo de cada capítulo deste livro. Códigos procedurais não são fáceis de serem reutilizados, tendem a não ser coesos, e flexibilização ocorre por meio de mais código escrito no mesmo lugar, tornando o código ainda mais complexo.

O interessante é que se você analisar com cuidado, alguns padrões de projeto (que são sempre bons exemplos de código OO) separam dados de comportamento. É o caso de *State*, *Strategy*, e muitos outros padrões. Esses são casos particulares, em que optamos por desacoplá-los para ganhar em flexibilidade. São decisões pontuais ao longo do sistema. Não é o caso de sistemas

anêmicos, onde isso acontece o tempo todo.

Você realmente precisa de um bom motivo para programar dessa maneira. Infelizmente é comum encontrarmos desenvolvedores que ainda defendem esse tipo de arquitetura. Ela é, na maioria dos casos, um grande passo para trás. Lembre-se: OO é a sua grande ferramenta. Nunca fuja dela. É fácil programar procedural em linguagens OO. Mais do que deveria ser.

5.10 Conclusão

Neste capítulo, discutimos o terceiro pilar de todo código bem orientado a objetos: encapsulamento. Esconda os detalhes da implementação, e diminua pontos de mudança. É isso que tornará seu sistema fácil de ser mantido.

Lembre-se que precisamos sempre diminuir a quantidade de pontos de mudança. Quanto menos, mais fácil. Quanto mais claras, melhor. Pense no seu sistema agora como um daqueles grandes abajures, em que, se você tocar em uma das partes, as partes mais abaixo balançam. Seu sistema deve ser idêntico: se você tocar em uma classe, você precisa ver facilmente as outras classes que deverão ser alteradas. E, claro, quanto menos, melhor.

No fim, não é tão difícil quanto parece.

Capítulo 6

Herança x composição e o tal do LSP

Já dizia Joshua Bloch: *"Crie suas classes pensando em herança, ou então proíba-a*

. Herança é sempre um assunto delicado. No começo das linguagens orientadas a objeto, a herança era a funcionalidade usada para vender a ideia. Afinal, reúso de código de maneira fácil, quem não queria? Mas, na prática, utilizar herança pode não ser tão simples. É fácil cair em armadilhas criadas por hierarquias de classes longas ou confusas.

Veja, por exemplo, a classe `ContaComum`:

```
public class ContaComum {

    protected double saldo;
```

```
    public ContaComum() {
        this.saldo = 0;
    }

    public void deposita(double valor) {
        if(valor <= 0)
            throw new ValorInvalidoException();

        this.saldo += valor;
    }

    public double getSaldo() {
        return saldo;
    }

}

    public void rende()      {
        this.saldo*= 1.1;
    }
}
```

Ela representa, de maneira simplificada, uma conta em um banco. A classe possui operações simples como `deposita()` e `rende()`.

Mas, como sempre, o sistema precisa crescer. Imagine agora a classe `ContaDeEstudante`, que é exatamente igual a uma conta, com a diferença de que ela não "rende". Usando herança, a implementação seria algo parecido com a que segue, onde o método `rende()` lança uma exceção:

```
public class ContaDeEstudante extends ContaComum {

    public void rende()  {
        throw new ContaNaoRendeException();
    }
}
```

Você já sobrescreveu métodos da maneira como acabamos de fazer?

6.1 Qual o problema dela?

É difícil enxergar o problema dessa simples sobrescrita. Para isso, imagine um código que faz uso de ambas `ContaComum` e `ContaDeEstudante`:

```
public class ProcessadorDeInvestimentos {

    public static void main(String[] args) {

        for (ContaComum conta : contasDoBanco()) {
            conta.rende();

            System.out.println("Novo Saldo:");
            System.out.println(conta.getSaldo());
        }
    }
}
```

O método `contasDoBanco()` retorna uma lista com diferentes contas. Não sabemos exatamente quais estão lá dentro, mas, dado o polimorfismo, podemos tratar todas elas pela referência da classe pai.

Agora o problema: qual o comportamento da aplicação? Não sabemos. Afinal, se houver alguma conta de estudante nesse código, a execução do programa parará, pois uma exceção será lançada. Pense que o sistema possui diversos desses loops e classes que interagem bem com `ContaComum` (e, por consequência, interagem com qualquer filho dela também). A nova classe `ContaDeEstudante` pode fazer essas classes pararem de funcionar também.

Por que isso aconteceu? Porque a classe filha quebrou o contrato (em Java, informal) definido pela classe pai: o método `rende()` na classe pai não lança exceção. Ou seja, as classes clientes não vão esperar que isso aconteça, e não vão tratar essa possibilidade.

Note que as **classes filhas precisam respeitar os contratos definidos pela classe pai**. Mudar esses contratos pode ser perigoso.

6.2 LSP Liskov Substitutive Principle

Para usar herança de maneira adequada, o desenvolvedor deve pensar o tempo todos nas pré e pós-condições que a classe pai definiu.

Toda classe ou método tem as suas pré e pós-condições. Por pré-condições, entenda os dados que chegam nela. Quais são as restrições iniciais para que aquele método funcione corretamente? Por exemplo, o método `deposita()` deve receber um inteiro maior que zero. Ou seja, o valor "1" é válido. Se uma classe filha de `ContaComum` mudar essa precondição para somente números maiores que 10, por exemplo, poderemos ter problemas.

As pós-condições são o outro lado da moeda. O que aquele comportamento devolve? O método `rende()` não devolve nada e não lança exceção. No exemplo que demos, já a classe `ContaDeEstudante`, esse mesmo método lança uma exceção. Problema.

Podemos sim mudar as pré e pós-condições, mas com regras. A classe filho só pode afrouxar a precondição. Pense no caso em que a classe pai tem um método que recebe inteiros de 1 a 100. A classe filho pode sobrescrever esse método e permitir o método a receber inteiros de 1 a 200. Veja que, dessa forma, todo o código que já fazia uso da classe pai continua funcionando.

Ao contrário, a pós-condição só pode ser apertada; ela nunca pode afrouxar. Pense em um método que devolve um inteiro, de 1 a 100. As classes que a usam entendem isso. A classe filho sobrescreve o método e devolve números só entre 1 a 50. Os clientes continuarão a funcionar, afinal eles já entendiam saídas entre 1 e 50.

Ou seja, não podemos nunca apertar uma precondição, e nem afrouxar uma pós-condição. É sobre isso que o Princípio de Substituição de Liskov discute. Ao herdar, você deve sempre lembrar do contrato estabelecido pelo classe pai. Apesar de parecer simples, na prática é complicado fazer esse tipo de análise em tempo de desenvolvimento.

6.3 O exemplo do quadrado e retângulo

Um exemplo bastante comum nas primeiras aulas de herança é o exemplo entre o `Quadrado` e o `Retângulo`. Afinal, eles são parecidos. Um quadrado é só um tipo especial de retângulo: parece o cenário perfeito para se usar herança e reaproveitar código para ambas as classes.

Imagine uma simples classe `Retangulo`, que armazena apenas o tamanho dos seus dois lados. A classe `Retangulo` poderia ser facilmente

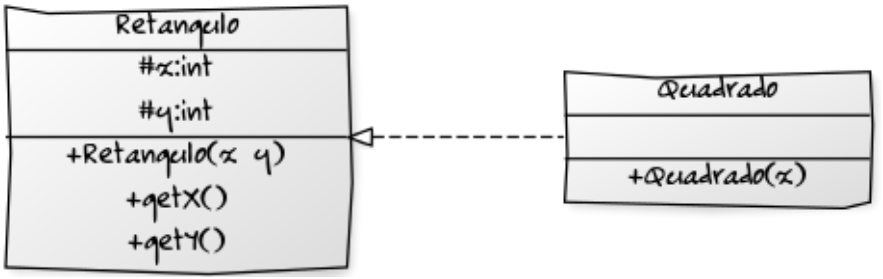
representada pelo código a seguir:

```
class Retangulo {
    private int x;
    private int y;

    public Retangulo(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() {
        return this.x;
    }

    return int getY() {
        return this.y;
    }
}
```

E o quadrado, por ser um retângulo "diferente", poderia ser uma extensão da classe `Retangulo`. O construtor da classe `Quadrado` passaria ambos os lados iguais para o construtor da classe pai:

```
class Quadrado extends Retangulo {
    public Quadrado(int x) {
        super(x,x);
    }
}
```

Perceba que a precondição da classe `Quadrado` é **mais forte** que a da classe filho. Em um quadrado, ambos os lados precisam ser iguais. Em um retângulo, não. De acordo com o princípio de Liskov, não poderíamos fazer essa herança. Claro, no exemplo do quadrado é difícil de se imaginar uma situação onde uma classe cliente, usando `Retangulo` como cliente, pudesse gerar um problema. Mas é um exemplo bastante didático para se perceber que herança não é fácil.



6.4 Acoplamento entre a classe pai e a classe filho

A discussão de acoplamento deixou isso claro. Sempre que uma classe depende da outra para existir, é acoplamento. E, dependendo da forma com que esse acoplamento é feito, podemos ter problemas no futuro.

É fácil perceber que a classe filho é totalmente acoplada à classe pai. Afinal, qualquer mudança no pai impacta no filho. Mas será que esse tipo de acoplamento pode ser problemático? Sem dúvidas. Problemas como esse podem inclusive ser vistos dentro da própria API do Java. O desenvolvedor que escrever uma *servlet*, por exemplo, precisa sobrescrever algum dos métodos para que ela funcione. Ele pode sobrescrever o método `service()` para que aquela *servlet* responda a qualquer tipo de requisição. Mas, se o desenvolvedor invocar o método `super.service()`, ele terá problemas, afinal a implementação do método pai lança uma exceção. Como a classe filho sabe disso? Não sabe. Sem o desenvolvedor olhar o código-fonte ou previamente conhecer o comportamento da classe, não há como saber.

```
public class MinhaServlet extends HttpServlet {
    public void service(HttpServletRequest req,
    HttpServletResponse res) {
        // se chamar o pai, a servlet nao funcionará
        super.service(req, res);
    }
}
```

O contrário também acontece. Na mesma *servlet*, se você sobrescrever o método `init()`, que é executado quando a *servlet* é carregada pelo servidor, e não invocar a implementação do pai, sua *servlet* não funcionará também. A implementação do pai é vital para ela. Como o desenvolvedor sabe disso? Novamente, se não conhecer a classe os detalhes da classe pai, ele não sabe.

```
public class MinhaServlet extends HttpServlet {
    public void init(ServletConfig cfg) {
        // chamar o método do pai é obrigatório
        super.init(cfg);
    }
}
```

Por isso, ao pensar em modelar hierarquias de classe usando herança, lembre-se do acoplamento entre classe pai e filho. Modele hierarquias nas quais as classes filhos precisam conhecer pouco (ou não conhecer nada) dos detalhes da classe pai.

Repare que isso não é fácil. Imagine uma classe `Funcionario`, que guarda o salário dele, e possui o cálculo de bonificação de fim do ano:

```
class Funcionario {
    // outros atributos aqui
    protected double salario;

    public double bonus() {
        return this.salario * 0.2;
    }

    // outros comportamentos aqui
}
```

Agora imagine uma classe filho de `Funcionario`, por exemplo, a classe `Gerente`, que é um funcionário, mas cujo cálculo de bônus é de 30% em vez de 20%. Uma solução apenas didática seria invocar o método do pai (que dá 20%) e somar mais 10%:

```
class Gerente extends Funcionario {
    private String placaDoCarro;
```

```
    public double bonus() {
        return super.bonus() +
          this.salario * 0.1;
    }

}
```

Apesar de funcionar, qualquer mudança na implementação do pai afetará diretamente o filho. E às vezes isso não é o desejado. Gerentes ganham 30%, independente dos outros funcionários. O ideal seria o método ter a sua própria implementação, sem depender da implementação do pai:

```
public double bonus() {
    return this.salario * 0.3;
}
```

Mas nunca devo usar o método do pai? Claro que sim, se fizer sentido. Se a regra fosse diferente: o bônus do gerente é igual ao bônus do funcionário mais R$ 500, faria todo o sentido usar a implementação do pai:

```
public double bonus() {
    return super.bonus() + 500;
}
```

Lembre-se, portanto, de tentar ao máximo reduzir o acoplamento entre a classe pai e a classe filho. Veja que podemos também usar o termo encapsulamento. Afinal, se a classe filho conhece demais da implementação da classe pai, é porque ela não encapsulou bem seus detalhes de implementação. E já sabemos os problemas de falta de encapsulamento.

Além disso, uma boa maneira para diminuir a chance de quebrar o encapsulamento é evitar o uso de `protected`. Sabemos que atributos das classes devem ser `private` para evitar que classes externas alterem seus dados sem qualquer controle. Com o `protected` é a mesma coisa. Queremos mesmo permitir que a classe filho altere atributos da classe pai sem qualquer tipo de controle? Se a resposta for não, talvez então valha a pena diminuir a visibilidade do atributo e prover métodos (`protected`, por exemplo) que os manipulem.

No exemplo da classe `Funcionario`, o atributo `salario` é `protected`. Isso permite que as classes filhas alterem o salário sem qualquer tipo de restrição. Se a regra de alteração de salário fosse composta por regras mais complicadas, encapsulá-las seria uma boa ideia.

6.5 Favoreça a composição

Usar herança é sim complicado, e o seu mau uso pode trazer problemas. É por isso que muitos desenvolvedores sugerem o uso de composição em vez de herança. Lembre-se de que você não usa herança apenas para "ganhar métodos na sua classe". Se você precisa usar a classe `Matematica`, análoga à classe `Math` do Java, dentro de uma classe sua de regra de negócio, não faz sentido usar herança. Use composição.

```
class CalculoDeImposto extends Matematica {
    // As regras de negócio da calculadora de imposto
    // fazem uso dos métodos da classe Matematica.
    // Usamos herança só para que o código ficasse
    // "menor".
}
```

A composição tem vantagens. A relação da classe principal com a classe dependida não é tão íntima quanto a relação existente entre classes pai e filho, portanto, quebrar o encapsulamento é mais difícil.

A composição nos dá flexibilidade. Os próprios padrões de projeto do GoF usam, em sua maioria, composição para trazer flexibilidade. Poucos deles usam herança para reutilizar ou estender comportamentos. É fácil trocar a implementação passada e obter um novo comportamento. Um bom exemplo é o que usamos para fazer comparações em Java. Podemos fazer a classe implementar a interface *Comparable* (não é herança, mas vale a discussão mesmo assim) e escrever o método `compareTo()`. Dessa forma, temos uma maneira de ordenar lista de elementos daquela classe. Mas temos apenas **uma** maneira. Ao passo que, se optarmos por criar diversas implementações de *Comparator*, podemos passar diferentes algoritmos de ordenação. Generalize esse exemplo agora para a herança. Ao optar por composição, ganhamos flexibilidade.

Escrever testes automatizados também é mais fácil. Mockar objetos e comportamentos e passá-los para classes que as usam para compor o comportamento é natural; com herança, muito mais difícil. É provavelmente por isso que códigos produzidos com TDD fazem menos uso de herança.

Em nosso exemplo das classes `ContaComum` e `ContaDeEstudante`, onde a manipulação de saldo é basicamente a funcionalidade que muda em cada classe filho, é possível extrair uma classe cuja única responsabilidade é manipulá-lo. As classes de `Conta`, por sua vez, fazem uso dela:

```
class ManipuladorDeSaldo {
    private double saldo;

    public void adiciona(double valor) { ... }
    public void retira(double valor) { ... }
    public void juros(double taxa) { ... }
    public double getSaldo() { ... }
}

class ContaComum {
    private ManipuladorDeSaldo manipulador;

    public ContaComum() {
        this.manipulador = new ManipuladorDeSaldo();
    }

    public void saca(double valor) {
        manipulador.adiciona(valor);
    }

    public void rende() {
        manipulador.juros(0.1);
    }
}

class ContaDeEstudante {
    private ManipuladorDeSaldo manipulador;

    public ContaDeEstudante() {
```

```
        this.manipulador = new ManipuladorDeSaldo();
    }

    public void saca(double valor) {
        manipulador.adiciona(valor);
    }

    // nao tem o metodo rende
}
```

Repare que a classe `ManipuladorDeSaldo` encapsula bem todo o comportamento, e ambas as classes apenas a utilizam. O problema dessa abordagem é a quantidade de métodos que apenas repassam a invocação de método para a classe utilizada. Veja que as implementações de `saca()` basicamente repassam a chamada para o método `adiciona()`. Como sempre, é uma troca. Com frequência, a melhor.

6.6 Herança para DSLs e afins

É comum códigos (em especial, códigos de teste) que fazem uso de herança apenas para facilitar a vida das classes filhas. Apesar de parecer uma má ideia e ir contra o que discutimos aqui, imagine uma classe que usa Selenium para escrever um teste de sistema. Obviamente essa classe faz uso da API do Selenium, que é por si só bastante verbosa. O trecho de código a seguir exemplifica:

```
class TesteDoCadastro {
    // driver do selenium
    private WebDriver driver;

    public TesteDoCadastro(WebDriver driver) {
        this.driver = driver;
    }

    public void testaCadastro() {
        driver.findElement(By.id("nome"))
            .sendKeys("Mauricio");
        driver.findElement(By.id("endereco"))
```

```
            .sendKeys("Rua Vergueiro");

        driver.findElement(By.id("btnSubmete")).click();

        String resultado = driver
            .findElement(By.id("resultado"))
            .getText();
        Assert.assertEquals("ok", resultado);
    }
}
```

A API do Selenium é verbosa e essa classe faz uso intenso dela. Veja quantas invocações para `findElement`, `sendKeys`, e assim por diante. E veja que elas se repetem: o `findElement()` recebe ainda o resultado de `By.id()`. Todo esse código poderia ser isolado. Por exemplo:

```
class TesteComSelenium {

    protected WebDriver driver;

    public TesteComSelenium(WebDriver driver) {
        this.driver = driver;
    }

    public void preenche(String id, String valor) {
        driver.findElement(By.id(id)).sendKeys(valor);
    }

    public void submete(String id) {
        driver.findElement(By.id(id)).click();
    }

    public String conteudo(String id) {
        return driver.findElement(By.id(id)).getText();
    }
}
```

Já que `TesteDoCadastro` **é um** `TesteComSelenium`, podemos usar herança para reaproveitar esses métodos:

```
class TesteDoCadastro extends TesteDoSelenium{

    public TesteDoCadastro(WebDriver driver) {
        super(driver);
    }

    public void testaCadastro() {
        preenche("nome", "Mauricio");
        preenche("endereco", "Rua Vergueiro");

        submete("btnSubmete");

        String resultado = conteudo("resultado");
        Assert.assertEquals("ok", resultado);
    }
}
```

Repare que o código ficou bem mais enxuto e fácil de ser lido. Apesar de a relação entre ambas as classes ser uma relação de herança justa, às vezes abrimos mão de boas práticas para facilitar a escrita e uso de DSLs. Logo, leve isso em conta: às vezes fazemos uma "má herança" para termos outros ganhos. Tenha isso em mente também.

6.7 Quando usar herança então?

Lembre-se da frase do Joshua no começo do capítulo: ou você modela a classe para usar herança, ou não. Herança deve ser usada quando existe realmente a relação de X **é um** Y. Por exemplo, `Gerente` é um `Funcionário`, ou `ICMS` é um `Imposto`. Não use herança caso a relação seja de composição, ou seja, X **tem um** Y, ou X **faz uso de** Y.

Um conjunto de classes que fazem bom uso de herança seguem os princípios discutidos anteriormente. Eles evitam ao máximo que a classe filho conheça detalhes da implementação do pai, e não violam as restrições de pré e pós-condições na hora de sobrescrever um determinado comportamento. Nada também o impede de usar herança e composição. Use herança para reaproveitar código que realmente faz sentido, e composição para trechos de código que precisam de mais flexibilidade.

Não saia deste capítulo pensando que herança deve ser evitada 100% das vezes. É só que ela é mais difícil de ser bem usada do que parece. Se você gerenciar bem todos os conceitos discutidos aqui (acoplamento, coesão e restrições entre classes pai e filhos), a herança será bem-vinda.

6.8 Pacotes: como usá-los?

Pacotes são a maneira que temos para agrupar classes que são parecidas. Lembre-se que o que está no mesmo pacote deve ser focado e suas classes sempre reutilizadas juntas. Também não se deve haver ciclos entre pacotes, ou seja, se o pacote `A` depende de `B`, então o pacote `B` não pode depender de `A`.

Você também pode pensar neles de maneiras diferentes. Muitas vezes temos subpacotes apenas para separar mais facilmente cada parte do módulo. Se temos um pacote denominado `metricas`, com subpacotes como `metrica1`, `metrica2` e `metrica3`, podemos enxergar todo o pacote como um só, o `metricas` (englobando os subpacotes). Pensando dessa forma, conseguimos reutilizar todo ele.

Lembre-se, a melhor regra é: deixe perto coisas que se relacionam e mudam juntas. Robert Martin, em seu trabalho, elencou alguns princípios de design de pacotes. É uma boa leitura.

6.9 Conclusão

Apesar dos extremistas afirmarem que herança deve ser evitada ao máximo, e este capítulo retratar mais os problemas do que as vantagens dela, espero que você tenha percebido que ela é um excelente recurso de linguagens orientadas a objetos, mas que deve ser usada com parcimônia. É muito fácil escrever código com problemas de acoplamento e coesão quando a usamos.

Não descarte herança, apenas favoreça a composição.

Capítulo 7

Interfaces magras e o tal do ISP

Vimos anteriormente que coesão é fundamental para manutenção e reúso de nosso código. Mas até então, discutimos coesão em nível de classe. Mas e interfaces? Elas precisam ser coesas? O que significaria uma interface coesa? Quais as vantagens?

Veja só a interface a seguir, responsável por calcular um imposto e gerar uma `NotaFiscal`:

```
interface Imposto {
    NotaFiscal geraNota();
    double imposto(double valorCheio);
}
```

Imagine agora uma primeira implementação dessa calculadora, para o imposto ISS, na qual o valor é 10% do valor cheio e a nota fiscal é gerada com os dados desse imposto:

```
class ISS implements Imposto {
    public double imposto(double valorCheio) {
        return 0.1 * valorCheio;
    }

    public NotaFiscal geraNota() {
        return new NotaFiscal(
            "Alguma informacao aqui",
            "Alguma outra informacao aqui"
        );
    }
}
```

Agora imagine um novo imposto, chamado IXMX, que é calculado também sobre o valor cheio, mas não emite nota fiscal. Como implementar a classe concreta? O que fazer com o método `geraNota()`? Podemos lançar uma exceção ou retornar um valor nulo, por exemplo:

```
class IXMX implements Imposto {
    public double imposto(double valorCheio) {
        return 0.2 * valorCheio;
    }

    public NotaFiscal geraNota() {
        // lanca uma exceção
        throw new NaoGeraNotaException();
        // ou retornar nulo
        return null;
    }
}
```

Sabemos que isso não é uma boa ideia, afinal isso mudará o contrato da classe `Imposto` e, por consequência, pode quebrar as classes clientes. Vimos isso quando estudamos princípio de Liskov. Repare que essa interface "gorda" (do inglês, *fat interface*) pode não ser a melhor em todos os casos. Isso faz sentido, afinal ela tem duas responsabilidades: calcular o valor do imposto e gerar uma nota.

7.1 Interfaces coesas e magras

A solução para o problema é análoga ao que tomamos quando discutimos classes coesas. Se uma classe não é coesa, dividimo-la em duas ou mais classes; se uma interface não é coesa, também a dividimos em duas ou mais interfaces. Veja:

```
interface CalculadorDeImposto {
    double imposto(double valorCheio);
}

interface GeradorDeNota {
    NotaFical geraNota();
}
```

Dessa forma, cada classe filha implementa quais interfaces forem necessárias, sem precisar fazer gambiarras ou coisa do tipo para se adequar a uma interface que não faz sentido para ela:

```
class ISS implements CalculadorDeImposto, GeradorDeNota {
    // os dois métodos aqui
}

class IXMX implements CalculadorDeImposto {
    // só implememenenta uma interface, pois
    // esse aqui não gera nota fiscal
}
```

Não há a necessidade de repetir todo o discurso aqui. Todo o capítulo de coesão em classes serve aqui. Lembre-se de fugir de interfaces gordas. Elas têm baixo reúso, e quando o desenvolvedor não tem a experiência necessária para perceber e resolver o problema, ele acaba por complicar ainda mais o projeto de classes.

Portanto, interfaces coesas são aquelas que possuem também apenas uma única responsabilidade. Quando coesas, essas interfaces possibilitam um maior reúso, tendem a ser mais estáveis (e sabemos que isso é bom!) e impedem "gambiarras" (como a mostrada na seção anterior, em que a classe não quer implementar um dos métodos da interface) de acontecerem. Sim, devemos pensar em coesão o tempo todo e em tudo que escrevemos.

7.2 Pensando na interface mais magra possível

Um dilema interessante sempre é: qual parâmetro devo receber em meu método? Por exemplo, imagine uma classe `NotaFiscal`, que é extremamente complexa e difícil de ser criada, pois depende de muitos outros objetos:

```
class NotaFiscal {
    public NotaFiscal(
        Cliente cliente,
        List<Item> itens,
        List<Desconto> descontos,
        Endereco entrega,
        Endereco cobranca,
        FormaDePagamento pagto,
        double valorTotal
    ) { ...}

    // muitos atributos e métodos
}
```

Agora, imagine um outro método, responsável por calcular o valor do imposto dessa nota fiscal. Mas, para calcular o valor, o algoritmo leva em consideração apenas a lista de itens. Os outros atributos não são necessários para ele. Observe:

```
class CalculadorDeImposto {
    public double calcula(NotaFiscal nf) {
        double total = 0;
        for(Item item : nf.getItens()) {
            if(item.getValor()>1000)
                total+= item.getValor() * 0.02;
            else
                total+= item.getValor() * 0.01;
        }
        return total;
    }
}
```

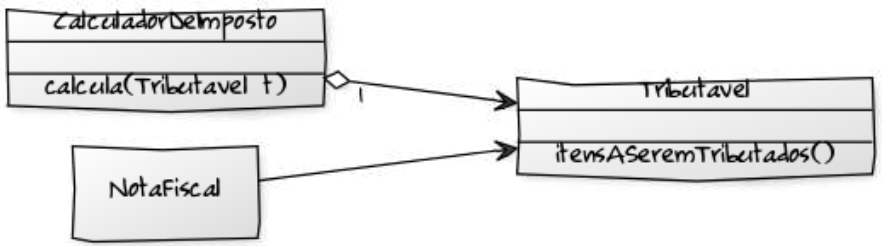
Veja só que o método `calcula()` recebe uma `NotaFiscal`. Mas veja que não precisaria. Poderíamos apenas receber uma `List<Item>`. A per-

gunta é: qual dos dois devemos receber? Essa é uma pergunta profunda, que envolve até uma discussão sobre acoplamento.

Discutimos anteriormente que devemos sempre nos acoplar com dependências mais leves e menos frágeis. Nesse exemplo em particular, `NotaFiscal` parece ser uma classe pouco estável e bastante complicada; já uma lista de itens parece ser mais simples. Receber essa lista como parâmetro pode deixar o método `calcula()` mais simples e menos propenso a sofrer modificações. Se, para o cálculo do imposto, o algoritmo precisasse apenas do valor total, por que não receber um `double total` apenas? Quanto mais simples, menos problemas.

Mas talvez uma lista de itens ou um *double* não deixe bem claro o que o método `calcula()` precisa receber. Um *double* pode ser qualquer número, uma lista de itens pode ser qualquer lista de itens, mas não é isso que ele quer: ele quer um valor de uma nota fiscal, ou uma lista de itens de uma nota fiscal. Garantias essas que tínhamos quando passávamos uma `NotaFiscal` como parâmetro.

Para resolver isso de maneira elegante, por que não criar uma abstração? Poderíamos ter uma interface chamada `Tributavel`, que seria implementada pela classe `NotaFiscal`. O método `calcula()`, por sua vez, receberia essa interface.



Assim como no exemplo anterior, acabamos emergindo uma interface leve. Nesse exemplo em particular, a criação da interface `Tributavel` nos traz vantagens:

- Diminuímos o risco do acoplamento, pois a interface `Tributavel` é muito mais estável que a classe `NotaFiscal`.
- O método `calcula()` recebe agora apenas o que realmente precisa. `Tributavel` é uma interface magra.
- Temos semântica no parâmetro do método. Não recebemos uma lista ou um *double* qualquer, e sim algo passível de tributação.

Em código:

```
interface Tributavel {
    List<Item> itensASeremTributados();
}

class NotaFiscal implements Tributavel {
    public NotaFiscal(...) { ... }

    public List<Item> itensASeremTributados() { ... }
}

class CalculadorDeImposto {
    public double calcula(Tributavel t) {
        double total = 0;
        for(Item item : t.itensASeremTributados()) {
            if(item.getValor()>1000)
                total+= item.getValor() * 0.02;
            else
                total+= item.getValor() * 0.01;
        }
        return total;
    }
}
```

Portanto, esse é mais um exemplo de como interfaces leves são importantes. Classes que dependem de interfaces leves sofrem menos com mudanças em outros pontos do sistema. Novamente, elas são pequenas, portanto, têm poucas razões para mudar.

7.3 E os tais dos repositórios do DDD?

No excelente livro *Domain-Driven Design* (DDD), do Eric Evans, ele comenta sobre os tais *repositórios*, que seriam abstrações para a camada de acesso a dados, com uma linguagem mais próxima do domínio da aplicação. Por exemplo, se você tem faturas em seu sistema, provavelmente teria uma interface e classe concreta como as que seguem:

```
interface RepositorioDeFaturas {
    List<Fatura> todas();
    void salva(Fatura f);
}

class FaturaDao implements RepositorioDeFaturas {
    // aqui usando hibernate ou qualquer coisa
    // do tipo para acessar os dados
}
```

A pergunta é: será que a interface é realmente necessária ou somente o DAO nos serviria bem? A resposta, como sempre, é: depende. Essa interface em particular nos seria bastante útil se tivéssemos a necessidade de mudar a maneira de acesso a dados. Mas, na prática, sabemos que raramente mudamos de banco de dados ou mesmo de framework de persistência. DAOs em particular são classes naturalmente estáveis, com interfaces geralmente bastante claras.

O principal aqui é lembrar de separar infraestrutura do resto, como já discutimos nos capítulos anteriores. Ter a interface ou não, nesse momento do livro, você já tem o conhecimento necessário para julgar se é necessário ou não.

7.4 Fábricas ou injeção de dependência?

Mas se nossas classes devem receber suas dependências pelo construtor, e assim ganhar em flexibilidade, quem instancia essas classes? Afinal, alguém precisa ir lá e dar o `new` no objeto. Por exemplo, no capítulo sobre classes abertas, tínhamos a classe `CalculadoraDePrecos`, e ela recebia

`TabelaDePreco` e `ServicoDeEntrega` em seu construtor. Mas quem instancia `CalculadoraDePrecos` ao longo do sistema?

Temos duas opções: usar algum framework de injeção de dependência, como Spring, Guice, entre outros, ou utilizar fábricas, isto é, classes cuja responsabilidade é instanciar outras classes. Se optássemos pela fábrica, teríamos uma classe parecida com a adiante. A única responsabilidade é criar a classe. E ela, sem ter muito como escapar, também instancia as dependências.

```
class FabricaDeCalculadoraDePrecos {

    public CalculadoraDePrecos constroi() {
        TabelaDePreco tabela = new TabelaDePrecosPadrao();
        ServicoDeEntrega tabela = new Correios();

        return new CalculadoraDePrecos(tabela, entrega);
    }
}
```

Obviamente, nada o impede de fazer um código ainda mais flexível, como, por exemplo, usando outra fábrica para instanciar uma dependência, ou mesmo lendo de algum arquivo de configuração:

```
Class<?> clazz = pegaTabelaDePrecoDaConfiguracao();
TabelaDePreco tabela = instanciaViaReflection(clazz);

// ...
```

A partir do momento em que você opta por usar uma fábrica, ela torna-se uma decisão importante do projeto. Você deve evitar instanciar a classe diretamente, e sempre usar a fábrica. Se sua linguagem suportar, você pode até usar os modificadores de visibilidade para fazer com que a classe não possa ser instanciada por outras classes, que não sua fábrica. E, se precisa ser ainda mais flexível, você pode fazer as classes ao redor do sistema receber a fábrica no construtor, em vez da classe concreta. Isso lhe possibilitará trocar inclusive a maneira na qual o objeto é criado.

Já frameworks de injeção de dependência também facilitam bastante sua vida. Ainda mais hoje, onde todos eles são de fácil configuração, e são bastante

flexíveis. As anotações do CDI, especificação Java para injeção de dependência, por exemplo, são bastante simples.

Se você está desenvolvendo um sistema maior, que já faz uso de outras bibliotecas, usar um framework de injeção de dependência talvez seja obrigatório. O VRaptor, por exemplo, framework MVC desenvolvido pela Caelum, dá suporte nativo a frameworks de injeção de dependência. Sem dúvida, tudo fica mais fácil.

Já a vantagem da fábrica é que ela independe de qualquer outra biblioteca para funcionar. Ela é uma solução OO simples e de fácil implementação. Muitos argumentam que a vantagem da fábrica em relação ao framework de DI é que o framework, por fazer tudo de forma automatizada e requerer menos código, "esconde" de você as suas reais dependências. Em uma fábrica, você precisa manualmente instanciar `C`, que é uma dependência de `B`, que, por sua vez, é uma dependência de `A`. Já o framework fará tudo isso automaticamente pra você, e a árvore de dependência fica implícita.

Não há a solução certa. Ambas têm vantagens e desvantagens. Você deve ver o que faz mais sentido para seu projeto.

Mas e o acoplamento da fábrica?

A fábrica é uma classe altamente acoplada, afinal, ela depende da classe e de todas as dependências que ela tem. Isso não faz dela uma má classe?

Sem dúvida, ela é altamente acoplada, mas na prática isso é menos problemático. Primeiro, porque fábricas tendem a ser classes estáveis; ela só quebrará quando a maneira de construir a classe principal mudar, e nesse caso, queremos mesmo que a fábrica pare de compilar. Segundo, ela é uma classe que não contém regras de negócio, portanto, precisamos "nos preocupar menos com ela". Terceiro, porque fábricas são decisões claras de design, e qualquer um que olhar pra ela sabe o seu papel no sistema.

Portanto, apesar de fábrica ser altamente acoplada, isso não é problema. Ela nos traz mais vantagens do que desvantagens.

7.5 Conclusão

Interfaces são fundamentais em bons sistemas orientados a objetos. Tomar conta delas é importante. Neste capítulo, discutimos as chamadas interfaces gordas, que são aquelas interfaces que contêm muitas responsabilidades diferentes. Assim como nas classes não coesas, essas interfaces também possuem baixo reúso e dificultam a manutenção.

Agora você percebeu que acoplamento, coesão, simplicidade e encapsulamento fazem sentido não só para classes concretas, mas sim para tudo.

Capítulo 8

Consistência, objetinhos e objetões

Discutir acoplamento, coesão, encapsulamento, classes abertas e extensíveis, interfaces magras etc., é realmente difícil. Mas se você chegou até aqui, fique tranquilo, pois o pior já passou. Neste capítulo, discutiremos agora outro conjunto de boas práticas, dessa vez relacionadas à consistência do objeto.

Preciso de objetos para representar um CPF, ou uma simples `String` resolve? Quais os problemas de ter objetos em estados inválidos? Como validar esse estado? Como garantir consistência dos objetos de domínio? São essas e outras questões que abordaremos aqui.

8.1 Construtores ricos

Imagine que você está lidando com uma classe `Pedido`, e na hora que você tenta persisti-la no banco de dados, ele reclama que não conseguiu salvar, pois a informação sobre o cliente estava nula; imagine que você foi executar um cálculo de imposto, mas o resultado deu zero, pois a taxa associada àquele imposto estava zerada. Sabemos que no mundo não existe pedido sem cliente, bem como imposto com valor zero (sim, isso é Brasil!).

Objetos em estado inválido são bastante problemáticos. Por estado inválido, entenda-se aquele objeto cujos atributos possuem valores não aceitáveis. Um pedido sem cliente, ou um cliente sem nome, são exemplos de objetos inválidos. O grande problema deles é que você não sabe o que esperar daquele objeto. Você não sabe se o comportamento que você acabou de invocar funcionará, pois provavelmente ele espera que o objeto esteja em um estado válido.

Veja, por exemplo, o trecho de código a seguir, que declara uma classe `Pedido`, e em seguida seta um a um seus atributos:

```
class Pedido {
    private Cliente cliente;
    private double valorTotal;
    private List<Item> itens;

    public void adicionaItem(Item it) {
        itens.add(it);
        recalculaValorTotal();
    }

    private void recalculaValorTotal() { ... }

    public double getValorTotal() { ... }

    public void setCliente(Cliente cliente) { ... }
}

// um cliente qualquer...
Pedido festa = new Pedido();
festa.adicionaItem(new Item("SALGADO", 50.0));
```

```
festa.adicionaItem(new Item("REFRIGERANTE", 50.0));
System.out.println(festa.getValorTotal());
```

Apesar de o método `adicionaItem()` encapsular corretamente a ideia de se adicionar um item ao pedido e automaticamente recalcular seu valor, veja que conseguimos instanciar um pedido sem cliente. Repare que o método `setCliente()` até existe e deve ser usado para armazenar o cliente. Mas agora depende de a classe cliente invocá-lo; se ela não invocar, o objeto estará em um estado inválido.

Garantir a integridade do seu estado é responsabilidade do próprio objeto. Ele não deve permitir que as classes clientes consigam levá-lo a um estado desse. Construtores são uma ótima maneira de se resolver esse problema. Se a classe possui atributos sem as quais ela não pode viver, eles devem ser pedidos no construtor. A classe deve exigi-los antes mesmo de ser instanciada. Veja, por exemplo, um bom construtor para a classe `Pedido`:

```
class Pedido {
    private Cliente cliente;
    private double valorTotal;
    private List<Item> itens;

    public Pedido(Cliente cliente) {
        // armazena o cliente
        this.cliente = cliente;

        // zera o valor total
        this.valorTotal = 0;

        // instancia a lista de itens
        this.itens = new ArrayList<Item>();
    }
}
```

Repare que agora é impossível criar um `Pedido` sem passar um `Cliente`. Podemos levar essa mesma ideia para ela. Se não faz sentido criar um cliente sem nome e telefone, devemos pedi-los no construtor:

```
class Cliente {
    private String nome;
```

```
    private String telefone;

    public Cliente(String nome, String telefone) {
        this.nome = nome;
        this.telefone = telefone;
    }
}
```

Veja um exemplo de uso de ambas as classes:

```
Cliente mauricio = new Cliente("Mauricio", "1234-5678");
Pedido festa = new Pedido(mauricio);
festa.adicionaItem(new Item("MUSICA AMBIENTE", 450.0));
```

Veja que agora todas as classes já nascem em estados válidos. Apesar de você não ter reparado, desde o primeiro exemplo, a classe `Item` já possuía um construtor mais rico.

Se você tem objetos que são complicados de serem criados e um simples construtor não resolve, você pode apelar para padrões de projeto criacionais, como é o caso do *Builder* ou da *Factory*. Builders ou Factories são nada mais do que classes que criam o objeto de que você precisa, de forma a garantir que seu estado esteja consistente.

> **Consistência sempre**
>
> Lembre-se também que a própria classe deve garantir que fique em um estado válido para sempre. Repare no atributo `valorTotal` da classe `Pedido`. Não é qualquer um que pode alterar o valor dela, afinal a regra para saber o valor total deve estar encapsulada na classe. Portanto, um `setValorTotal()` quebraria o encapsulamento e provavelmente seu uso levaria a classe a um estado inválido.

Outro bom exemplo é o caso de atributos que são necessários, mas que é possível prover um valor padrão para ele. Imagine uma classe `Carro`, que tem como atributos `Pneu` e `Motor`. Todo carro tem pneu e motor, mas você é capaz de prover um pneu e um motor padrões. Nesse caso, por que não dois construtores, sendo que um deles provê os valores padrão?

```
class Carro {
    private Pneu pneu;
    private Motor motor;

    public Carro(Pneu pneu, Motor motor) {
        this.pneu = pneu;
        this.motor = motor;
    }

    public Carro() {
        this(new PneuPadrao(), new MotorPadrao());
    }
}
```

Garanta consistência dos seus objetos. Use construtores. Novamente, se seu objeto possui atributos necessários desde o primeiro momento, não permita que o objeto seja criado sem eles.

Construtores para atender frameworks

Alguns frameworks não lidam bem com a ideia de a classe ter construtores ricos e nenhum construtor padrão. O Hibernate, framework ORM bastante popular de mercado, é um exemplo. Nesse caso, a sugestão é que você tenha construtores ricos, e também um construtor padrão, com a menor visibilidade possível, marcado como *deprecated*. Dessa forma, o framework conseguirá utilizá-lo, mas os desenvolvedores são avisados de que seu uso deve ser evitado.

8.2 Validando dados

Capturar dados do usuário é sempre uma tarefa interessante. Usuários (ainda mais quando leigos) tendem a informar os mais inusitados valores para nossos programas. É comum vermos, por exemplo, alguém digitando um número em um campo que deveria receber uma data de nascimento, ou uma string em um campo que deveria receber um valor numérico.

É, portanto, responsabilidade de nossos sistemas tratar esse tipo de situação. Agora a pergunta é: onde? A resposta, como sempre, é "depende". Antes de começar a discussão, vamos separar a validação em dois tipos diferentes: primeiro, aquele conjunto de validações para garantir que o tipo de dado enviado pelo usuário seja válido. Ou seja, o campo "Idade" passado pelo usuário é um número, o campo "E-mail" é um e-mail, e assim por diante. Em segundo lugar, validações de negócio, ou seja, que determinado imposto precisa ser maior que 1%, ou que pessoas físicas precisam de um CPF.

Para o primeiro caso, a boa ideia é mais natural. Sempre que você recebe dados do usuário, isso é feito em alguma camada intermediária. Em uma aplicação web, por exemplo, você os recebe no *Controller*. Idealmente, esse tipo de validação deve ser feito lá. Você não deve permitir que dados sujos cheguem ao seu domínio. É na verdade para isso que servem controladores: para fazer a ponte entre o mundo do usuário, onde ele interage com formulários etc., e o mundo do domínio, onde as regras vivem. Essa ponte deve se preocupar em só deixar passar coisas válidas de um lado para o outro.

Portanto, faça esse tipo de validação no controlador. Campos não preenchidos, campos preenchidos com dados problemáticos etc., podem ser capturados lá. Isso até fará com que seu código de domínio fique mais limpo, afinal esse tipo de código de tratamento costuma ser extenso e feio:

```
@Path("/pedido/novo")
public void salva(String carrinhoId, String codigoItem) {
    if(carrinhoId == null) {
        validacoes.add("Você deve estar em um carrinho válido");
    }
    if(codigoItem == null || codigoItem.isEmpty()) {
        validacoes.add("Você deve escolher um item válido");
    }

    if(validacoes.hasErrors()) {
        return erro();
    }

    // ...
}
```

Esse tipo de discussão é bastante comum em arquiteturas hexagonais. Projetistas que gostam dessa linha de raciocínio preferem deixar toda essa validação simples de dados nos "adaptadores". Já as regras de validação que envolvem negócio, dentro das "portas". Independente da nomenclatura, faz todo o sentido você garantir que o tráfego de dados entre um mundo e outro (imagine o mundo HTML e o mundo de classes de domínio) seja somente de dados que façam sentido. A entidade responsável por essa transferência deve então garantir que isso aconteça, validando os dados que passam por ela.

Agora, o segundo caso é muito mais complicado. Onde colocar as validações de negócio? Aquelas que são mais complicadas e são realmente contextuais? As alternativas são inúmeras, e a escolha depende muito da complexidade da regra.

Podemos, por exemplo, tratar os dados dentro da própria entidade. Imagine, por exemplo, uma classe `CPF`. Não é qualquer número que é um CPF válido. A própria classe `CPF` poderia ser então responsável por esse tipo de validação. Veja o código a seguir, por exemplo, onde o construtor da classe se responsabiliza por isso:

```
class CPF {
    private String cpf;

    public CPF(String possivelCpf) {
        // as regras para validar um CPF aqui

        if(regrasOk) {
            this.cpf = possivelCpf;
        } else {
            throw new IllegalArgumentException("CPF invalido");
        }
    }
}
```

Dessa maneira, a classe `CPF` só será instanciada se o parâmetro passado for válido. A vantagem dessa abordagem é que você garante que o objeto nunca terá um valor inválido, afinal o próprio construtor o recusa. A desvantagem é que muitas pessoas não gostam de tratar exceções que são disparadas pelo construtor. Apesar de possível, a sensação é que isso não é elegante. Se a

exceção for checada, você precisará explicitamente tratá-la, o que acaba por poluir o código.

Outra abordagem é você prover um método que diz se o objeto é válido ou não. Na classe `CPF`, por exemplo, seria algo como ter um método `valida()` que diz se a classe está em um estado válido ou não. Nesse caso, você, como projetista da classe, permite que o usuário instancie uma classe com um CPF inválido:

```
class CPF {
    private String cpf;

    public CPF(String cpf) {
        this.cpf = cpf;
    }

    public boolean valida() {
        // regras aqui

        if(valido) return true;
        return false;
    }
}
```

A implementação do método `valida()` fica a critério do projetista. O método pode optar por lançar exceção ou mesmo retornar uma lista de erros de validação (afinal, uma classe maior pode ter muito mais coisa para validar e diferentes erros podem acontecer).

Uma implementação intermediária seria a implementação de um *builder* para a classe CPF. Esse builder, por sua vez, seria responsável por validar o CPF passado e retornar o erro para o usuário. A vantagem do builder é que você garante que a classe `CPF` não será criada com um valor inválido, e ao mesmo tempo, não coloca o código de validação (que pode ser complexo) dentro da própria classe de domínio. Novamente, a decisão de como devolver os erros é do projetista. Ele pode optar por uma simples exceção, bem como retornar algum objeto rico com a lista de erros encontrados.

```
class CPFBuilder {
```

```
    public CPF build(String cpf) {
        // regras aqui

        if(regrasOk) return new CPF(cpf);
        throw new IllegalArgumentException("CPF invalido");
    }
}
```

Dependendo da complexidade e da necessidade de reúso dessa regra de validação, ela pode ser colocada em uma classe específica. Por exemplo, uma classe chamada `ValidadorDeCliente` pode fazer as diversas validações que uma classe `Cliente` contém, por exemplo, clientes menores de idade devem ter obrigatoriamente nome do pai e mãe, clientes com mais de 60 anos precisam do número de convênio etc.:

```
class ValidadorDeCliente {
    public boolean ehValido(Cliente cliente) {
        // regras aqui

        if(regrasOk) return true;
        return false;
    }
}

public void novoCliente(Cliente cliente) {

    if(validador.ehValido(cliente)) {
        // continua processamento
    } else {
        // exibe erro ao usuário
    }

}
```

Mas qual a vantagem de colocar essa regra de validação para fora e não usar o *builder*? A vantagem é que, se sua regra de validação for complexa, isso pode ajudar. Imagine que um cliente precise ter telefone em uma determinada parte do sistema, mas não obrigatoriamente em outra parte. Você pode passar a compor regras de validação. Uma implementação para isso seria usar

algo similar a um *decorator* ou um *chain of responsibility*. Veja no código a seguir, onde compomos a regra de validação, criando uma regra que obrigue o usuário a informar nome, telefone e CPF. Obviamente, essa implementação é a gosto do cliente, podendo devolver a lista de erros ocorridos, para que o desenvolvedor consiga mostrar uma mensagem mais amigável ao usuário.

```
public void novoCliente(Cliente cliente) {

    Validacao validador =
        new NomeRequerido(
        new TelefoneRequerido(
        new CPFRequerido(
    )));

    if(validador.ehValido(cliente)) {
        // continua processamento
    } else {
        // exibe erro ao usuário
    }

}
```

Como você pode ver, existem diversas abordagens diferentes para validação. Todas elas possuem vantagens e desvantagens. Lembre-se de procurar por qual se encaixa melhor em seu problema: se suas regras de validação são complexas, partir para uma solução mais flexível é necessário; se elas forem simples, talvez deixá-las puramente em seus controladores seja suficiente.

8.3 Teorema do bom vizinho e nulos para lá e para cá

Outro grande problema de aplicações é tratar o dado que veio de alguma outra classe cliente. Quantas vezes você já não fez algum tipo de *if* para garantir que o objeto recebido não era nulo? E pior, quantas vezes você já esqueceu de tratar se o objeto era nulo e isso fez sua aplicação não funcionar?

Mas a pergunta é: o que seria do mundo se ninguém nunca passasse nulo para sua classe? Veja que você, classe, não precisaria se preocupar mais em

garantir que o objeto recebido não é nulo. Essa é a ideia do teorema do bom vizinho. Bom vizinho no sentido de que você é educado e não passará dados inválidos para a outra classe.

Em aplicações convencionais (entenda-se aplicações que não são bibliotecas ou frameworks), essa é uma prática que faz todo o sentido. A classe cliente deve invocar métodos de outras classes sempre passando dados válidos para ela. Ou seja, todo mundo é responsável por minimamente tratar os dados antes de passá-los para a próxima classe.

Generalize a ideia. Tente ser amigável com a classe que você irá consumir. Faça bom uso da interface pública que ela provê. Use sempre a sobrecarga correta; muitas vezes a classe lhe dá a sobrecarga sem aquele parâmetro que você não tem em mãos naquele momento. Se você tem dados que realmente podem ser nulos, pense em usar *Null Objects*, ou algo parecido (algumas linguagens já têm os tais `Optional`, que forçam o programador a tratar se ele é nulo ou não).

Obviamente, em bibliotecas e frameworks, tratar o nulo vai fazer a diferença entre sua aplicação funcionar e não funcionar. Afinal, se você desenvolve um framework MVC, e o usuário por algum motivo passou nulo, o framework deve saber lidar com isso.

8.4 Tiny Types é uma boa ideia?

É engraçado o momento em que decidimos quando criar um novo tipo no sistema. É também engraçado o momento em que não decidimos criar um novo tipo. Quantas vezes você já viu CPF ser representado por uma `String` ou mesmo um Endereço ser representado por 4 ou 5 `strings` (rua, bairro, cidade etc.)? Por que não criamos um tipo particular para representar cada uma dessas pequenas responsabilidades que aparecem em nossos sistemas, como CPF, telefone ou data de nascimento?

Muitas vezes acreditamos que uma string é suficiente para representar um telefone. Mas o que ganharíamos se tivéssemos uma classe `Telefone`, simples e pequena?

```
class Telefone {
    private String telefone;
```

```
    public Telefone(String telefone) {
        // alguma possível validação
        this.telefone = telefone;
    }

    public String get() { ... }
}
```

Classes como essa, que representam uma pequena parte do nosso sistema e praticamente não têm comportamentos, são conhecidas por **tiny types**. Levar essa abordagem a sério pode ser interessante. Imagine um sistema onde todos os tipos são no mínimo *tiny types*. Construir um objeto seria algo como o seguinte. Veja que dessa forma não há a possibilidade de passarmos informações em lugares errados; isso é bastante comum quando temos um método cuja assinatura é um conjunto de `Strings`. Aqui, o primeiro sistema de tipos documenta o que aquele método recebe.

```
// sem tiny types, você precisa inferir o que significa aquela
// string
Aluno aluno = new Aluno(
    "Mauricio Aniche",
    "mauricioaniche@gmail.com",
    "Rua Vergueiro",
    "São Paulo",
    "11 1234-5678",
    "12345678901"
    );

// com tiny types, o próprio tipo deixa isso claro.
Aluno aluno = new Aluno(
    new Nome("Mauricio Aniche"),
    new Email("mauricioaniche@gmail.com"),
    new Endereco("Rua Vergueiro", "São Paulo"),
    new Telefone("11 1234-5678"),
    new CPF("12345678901")
);
```

Cada pequeno tipo pode também fazer sua própria validação, e dessa

maneira, garantir que todo o conteúdo seja válido. Reutilizar fica mais fácil; `Email` é provavelmente um tipo utilizado em muitos outros lugares do seu sistema.

A desvantagem é justamente a quantidade de código a mais que existirá no sistema. Criaremos muitas classes, o que implica em muito mais código. Às vezes, não precisamos de tamanha abstração. Em alguns contextos, uma simples `String` é suficiente para representar um e-mail.

Além disso, você precisa ver como o ecossistema ao redor do seu sistema atua com *tiny types*. Para usar Hibernate, por exemplo, você precisará fazer consultas SQL maiores. Versões mais antigas de JSPs e EL também fariam o desenvolvedor sofrer um pouco mais para conseguir usar.

No fim, a ideia é você sempre balancear. Tiny types são interessantes e nos ajudam a ter um código mais flexível, reusável e fácil de manter. Mas também podem complicar trechos de código que seriam por natureza simples.

8.5 DTOs do bem

Desenvolvedores odeiam DTOs; ponto final. Isso se deve muito ao passado negro deles. Afinal, na época escura do JavaEE, onde as pessoas seguiam todas as más práticas de desenvolvimento que eram propagadas pela Sun, como o uso de *business delegates* e o não uso de orientação a objetos, DTOs eram bastante comuns.

DTOs, como o próprio nome diz, são *data transfer objects*. São objetos que devem ser usados para transmitir informações de um lado para o outro. Eles geralmente não possuem comportamentos, e são apenas classes com atributos. A sensação é de que, quando temos classes como essas, estamos criando código que não é orientado a objetos; afinal códigos OO possuem dados e comportamento juntos.

Mas, em muitas situações, DTOs são importantes. Se você parar para pensar, temos diversos pontos do nosso sistema onde uma classe de negócio não está bem representada. Por exemplo, podemos dividir nossos sistemas em dois grandes módulos: a interface que o usuário vê e com a qual interage (por exemplo, o conjunto de HTMLs que ele vê em uma aplicação web) e o conjunto de regras de negócio (que estão encapsuladas em classes de domínio).

Nesse meio, temos algum trecho de código que converte a entrada do usuário no HTML em chamadas de objetos das classes de domínio; em aplicações web, *controllers* fazem bem esse papel.

Em muitos casos, a tela em que o usuário atua não é idêntica ao objeto de negócio. O que é comum fazermos na prática? Tentar forçar a interface a ter a mesma representação daquele objeto. Ou forçar nosso framework MVC a converter aquela tela no objeto de negócio, mesmo que ele deixe alguns campos em branco. A pergunta é: por que não criar um DTO que represente exatamente aquela interface do usuário? E aí, mais pra frente, o DTO é convertido em uma ou mais classes de domínio. Analogamente, você quer exibir dados para o usuário na interface, mas de forma diferente a qualquer uma das suas classes de negócio. Ou seja, por que não usar DTOs para realmente transferir dados entre camadas, mesmo que elas não sejam físicas?

Por que não criar uma classe que represente o que seu usuário realmente precisa ver naquela tela, criando por exemplo a classe `DadosDoUsuarioDTO`, em vez de ficar passando conjuntos de inteiros e strings separados e deixar a JSP mais complicada e menos semântica? Um exemplo análogo são relatórios. Muitos relatórios são formados a partir do agrupamento de muitos dados diferentes, portanto, não são bem representados por uma única interface.

```
class DadosDoUsuarioDTO {
    private String nome;
    private Calendar ultimoAcesso;
    private int qtdDeTentativas;

    public DadosDoUsuarioDTO(...) { ... }

    // getters e setters
}
```

Não tenha medo de criar DTOs que representem pedaços do seu sistema. Facilite a transferência de dados entre suas camadas; aumente a semântica deles. Lembre-se que o problema não é ter DTOs, mas sim só ter DTOs.

8.6 Imutabilidade x mutabilidade

Estamos acostumados com classes cujo estado é mutável, ou seja, podem mudar. As suas próprias classes de domínio contêm *setters*. Ou seja, você pode mudá-las a hora que quiser. Claramente, isso é uma vantagem, afinal, podemos mudar seu estado interno, e isso pode fazer com que ela se comporte diferente a partir daquele momento.

Mas, se o desenvolvedor não estiver atento, é fácil cometer um deslize e trabalhar com uma variável cujo valor atual não sabemos bem. A classe `Calendar`, do Java, por exemplo, quando você adiciona alguns dias nela, muda seu estado interno, para representar a nova data. Isso pode complicar o desenvolvedor. Veja que a variável, apesar de se chamar `hoje`, não contém mais a data de hoje:

```
// hoje terá a data de hoje
Calendar hoje = Calendar.getInstance();

// hoje agora terá 1 dia depois de hoje
// então, não é mais hoje!
hoje.add(Calendar.DAY, 1);
```

Alterar o estado da classe pode ser um problema justamente porque você não sabe se o estado dela foi alterado em algum lugar, então seu programa não está esperando uma alteração nela. Um outro exemplo comum, e já discutido no capítulo de encapsulamento, são classes de domínio que devolvem listas internas. Imagine uma classe `Pedido` que tem uma lista de `Item` e um *getter* para essa lista:

```
class Pedido {
    private List<Item> itens;

    public void adicionaItem(String produto) {
        // regra de negocio totalmente encapsulada aqui
        // para se adicionar um item
    }

    public List<Item> getItens() {
        return itens;
```

```
    }

}
```

A classe está bem escrita. Ela inclusive tem o método `adicionaItem()`, que encapsula toda a regra de negócio para se adicionar um item dentro desse pedido. O problema é que, como esse `getItens()` devolve a lista original, qualquer classe pode colocar novos itens nessa lista, sem passar pelas regras de negócio que um pedido tem.

É justamente por isso que muitos desenvolvedores defendem as famosas classes imutáveis. Ou seja, classes que, após instanciadas, não mudam nunca seu estado interno. Se uma classe não pode ser modificada, você não tem mais o problema de alguma outra classe perdida no sistema fazer uma modificação que você não está esperando e atrapalhar toda a sua lógica atual. Além disso, paralelizar fica mais fácil, pois não há escrita concorrente nesses objetos.

Escrever uma classe imutável não é complicado. Basta evitar o uso de *setters*, por exemplo. Ou, se você precisar dar um método que modifica o conteúdo do objeto, esse objeto deve devolver uma nova instância dessa classe, com o novo valor. Veja, por exemplo, a classe `Endereco` a seguir, imutável. Repare que, ao mudar a rua, ela devolve uma nova instância da classe:

```
class Endereco {

    private final String rua;
    private final int numero;

    public Endereco(String rua, int numero) {
        this.rua = rua;
        this.numero = numero;
    }

    // getRua e getNumero aqui

    public Endereco setRua(String novaRua) {
        return new Endereco(novaRua, numero);
    }
```

```
    public Endereco setNumero(int novoNumero) {
        return new Endereco(rua, novoNumero);
    }
}
```

Muitas APIs hoje seguem essa ideia. A classe `Calendar`, citada aqui, não segue, mas a biblioteca *Joda Time*, uma biblioteca famosa no mundo Java para manipular data e hora é totalmente imutável. Sempre que você muda algo na data, como dia, mês, ano, hora, ela devolve uma nova instância de `LocalDateTime`, com os novos dados.

Imutabilidade tem lá suas vantagens, mas também não é bala de prata. O projetista deve encontrar os melhores lugares para usar a técnica. Imagine o mundo real mesmo: algumas coisas não mudam nunca mesmo. Uma data não muda nunca (o dia 23/01/1986 será o mesmo pra sempre), um endereço não muda nunca (Rua Vergueiro, 3185, existirá pra sempre), portanto, são ótimas candidatas a serem classes imutáveis. Já um pedido ou um cliente são totalmente passíveis de mudança. Nesses casos, classes mutáveis podem ser uma boa ideia.

8.7 Classes que são feias por natureza

Queremos código limpo e bonito o tempo todo, claro. Mas algumas classes tendem a ser mais feias do que outras. Em particular, classes que servem de adaptadores, ou seja, conectam dois trechos diferentes da aplicação, tendem a ter códigos que dão orgulho a qualquer um. Pegue qualquer código de um controlador seu, por exemplo. Veja que ele é cheio de variáveis, *ifs* que verificam se os objetos não são nulos, depois instanciação de classes, invocação de métodos, captura de retornos, e por fim, a criação de uma estrutura de dados para mandar os dados que importam para a camada de visualização.

Algumas classes nasceram para serem feias. Controladores são um bom exemplo disso. O código deles é procedural, e será assim pra sempre. Fábricas também tendem a ter péssimos códigos, cheios de *ifs* e códigos que pegam configurações de arquivos. Triste, não!? Mas a moral desta seção é que isso não é um problema!

Código existe e existirá pra sempre. Sua missão é fazer com que esse

código feio esteja nas pontas da sua aplicação. Eles podem existir em classes que fazem a ponte entre dois mundos, em classes que escondem algum sistema legado, ou mesmo em classes que lidam diretamente com algum tipo de infraestrutura. Classes como essa possuem códigos feios, mas que são extremamente estáveis. Você raramente encosta em uma fábrica depois que ela está pronta, ou raramente encosta em uma classe que monta o JSON que será enviado para algum serviço web de terceiros. Ou seja, apesar de feias, elas não são propensas a gerarem bugs com frequência.

Diferentemente das suas classes de domínio, cujas regras mudam e evoluem constantemente. Ali, você deve focar todos seus esforços para que aquele código seja fácil de ser mantido. Código limpo sempre.

Ou seja, não tenha medo de ter código feio, contanto que ele esteja bem controlado, escondido, e que você não precise lembrar que ele existe o tempo todo.

8.8 Nomenclatura de métodos e variáveis

Nomenclatura é outro detalhe importante em sistemas OO. É por meio dos nomes que damos às coisas, como classes, pacotes, métodos e atributos, que conseguimos entender o significado de cada um deles. É impossível saber o que uma classe com o nome `Xpto` faz, ao passo que uma classe `NotaFiscal` deixa bem claro o que ela representa.

É difícil dar dicas para bons nomes. É algo extremamente particular. Por exemplo, qual nome é melhor para um método `qtdItens()` ou `quantidadeDeItens()`? Cada um terá sua opinião. O importante é que ambos são legíveis e de fácil compreensão.

Siga as convenções definidas pela sua equipe. Decidam se vão usar as convenções da sua linguagem. Em C#, por exemplo, é comum batizarmos interfaces com o prefixo "I", ou seja, `ITributavel` em vez de simplesmente `Tributavel`. Algumas equipes gostam, outras não. A vantagem da convenção da linguagem é que qualquer desenvolvedor do planeta a entende; mas se sua equipe não gosta, e tem uma razão para isso, não há motivos para segui-la.

Evite variáveis com nomes muito longos ou muito curtos. Uma variável

que se chama `x` não diz nada, ao passo que uma variável que se chama `todosOsImpostosCalculadosJuntosESomados` incomoda os olhos de qualquer um.

A regra é que não há regra. Apenas reflita bastante sobre os nomes escolhidos. Eles é que são responsáveis pela semântica do seu projeto de classes.

8.9 Conclusão

Neste capítulo, discuti outro conjunto de técnicas e boas práticas que são totalmente contextuais, ou seja, diferentes de acoplamento e coesão, que fazem sentido para todo tipo de classes. Validação, imutabilidade, *tiny types* etc. têm seu espaço e fazem sentido em muitos casos.

A ideia é você não parar por aqui e continuar a estudar o conjunto de boas práticas contextuais que existe para a combinação linguagem-framework-domínio em que você atua. Elas existem, e aos montes.

Capítulo 9

Maus cheiros de design

Agora que você tem em mãos um grande conjunto de princípios e boas práticas, e sabe como resolver os diferentes problemas que um código pode ter, está na hora de darmos nomes não só às boas práticas, mas também às más práticas. Conhecê-las também é fundamental. Vai ajudá-lo a identificar mais rapidamente os problemas, e até a se comunicar sobre eles.

Esse conjunto de "más práticas" é conhecido por *smells* (do inglês, mau cheiro). Uma rápida busca por *code smells* e você encontrará uma lista infindável deles. Neste capítulo, discutirei alguns deles, bastante comuns na literatura. Não tentarei traduzir os nomes desses maus cheiros, porque o nome em inglês é bastante popular. Leve este capítulo como uma curta revisão de tudo o que discutimos ao longo do livro.

9.1 Refused Bequest

O capítulo de herança foi, com certeza, um dos capítulos mais densos deste livro, com toda aquela discussão de prés e pós condições em que devemos pensar na hora de sobrescrever um método. **Refused Bequest** é o nome dado para quando herdamos de uma classe, mas não queremos fazer uso de alguns dos métodos herdados. Ou seja, a classe filho só precisa usar partes da classe pai.

Esse mau cheiro fica ainda pior quando a classe herda da classe pai, mas não quer, de forma alguma, ser tratada pela abstração. Por exemplo, se temos uma classe `NotaFiscal` que herda da classe `Matematica`, simplesmente porque ela tem métodos úteis para ela, se em algum lugar passarmos uma referência de nota fiscal para alguém que quer uma classe de matemática, ela provavelmente se comportará mal.

```
class Matematica {
    public int quadrado(int a, int b) {}
    public int raiz(int a) {}
}

class NotaFiscal extends Matematica {

    public double calculaImposto() {
        // alguma conta qualquer
        return quadrado(this.valor, this.qtd);
    }
}

// uma classe cliente qualquer
NotaFiscal nf = new NotaFiscal();
algumComportamento(nf);

public void algumComportamento(Matematica m) {
    // aqui pode vir uma nota fiscal, mas
    // não queremos uma nota fiscal...
    // queremos matematica!
}
```

Lembre-se que quando a classe filho herda de uma classe pai qualquer,

mas "não quer" isso, você tem um mau cheiro no seu projeto de classes.

9.2 Feature Envy

De nada adianta sabermos orientação a objetos, e separarmos dados de comportamentos. É para isso que criamos classes: para agrupar dados e comportamentos que manipulam esses dados. **Feature Envy** é o nome que damos para quando um método está mais interessado em outro objeto do que no objeto em que ele está inserido.

Veja o código a seguir. A classe `NotaFiscal` é uma entidade como qualquer outra, com atributos e comportamentos. A classe `Gerenciador` (um péssimo nome de classe...) faz uso dela. Mas é fácil ver que o método `processa()` poderia estar em qualquer outro lugar: ele não faz uso de nada que a sua classe tem. E ela faz diversas manipulações na nota fiscal. Esse código estaria muito melhor se estivesse dentro da classe `NotaFiscal`.

Esse é o típico código procedural. Módulos fazem uso intenso de dados que estão em outros lugares. Isso faz com que a manutenção de tudo relacionado à nota fiscal fique mais difícil. O desenvolvedor precisa saber todos os pontos que têm pequenas regras de negócio como essa. Na prática, ele nunca sabe.

```
class NotaFiscal {
    private double valor;
    private String cliente;
    private int qtdDeItens;

    // ...
}

class Gerenciador {
    private Usuario usuarioLogado;

    public void processa(NotaFiscal nf) {
        double imposto = nf.calculaImposto();
        if(nf.getQtdDeItens() > 2) {
            imposto = imposto * 1.1;
        }
```

```
        nf.setaValorImposto(imposto);

        nf.marcaDataDeGeracao();
        nf.finaliza();
    }
}
```

Claro, alguns métodos realmente precisam invocar métodos de muitas outras dependências, mas você deve ficar atento a códigos como esses. Às vezes, o comportamento está no lugar errado. Pergunte a você mesmo se ele não poderia estar dentro da classe que aquele método está usando.

9.3 Intimidade inapropriada

No capítulo de encapsulamento, também falamos de **intimidade inapropriada**. É algo bem parecido. Imagine outra classe conhecendo e/ou alterando detalhes internos da sua classe? Problemático e já sabemos o porquê: a falta de encapsulamento faz com que o desenvolvedor precise fazer a mesma alteração em diferentes pontos.

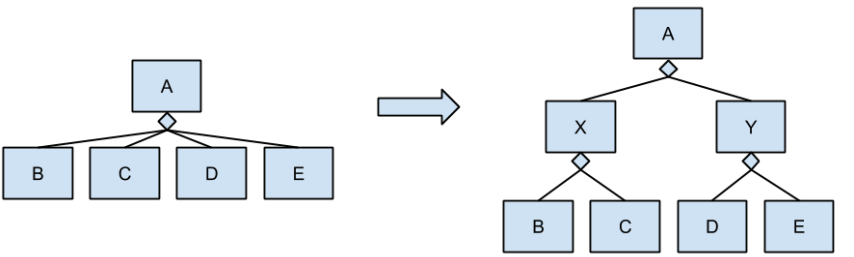
Veja o código a seguir, muito parecido com o anterior, onde um determinado método conhece demais sobre a implementação interna da classe `NotaFiscal`. Observe que o código faz perguntas para o objeto e, dada a resposta, ele decide marcá-la como importante. Essa é uma decisão que deveria estar dentro da nota fiscal. Imagine que em algum outro ponto do seu sistema você criará e manipulará notas fiscais. E se você criar uma nota fiscal, encerrá-la, for maior que 5000, e você esquecer de marcá-la como importante?

```
public void processa(NotaFiscal nf) {
    if(nf.isEncerrada() && nf.getValor() > 5000) {
        nf.marcaComoImportante();
    }
}
```

Procure sempre colocar comportamentos nos lugares corretos, e não deixe suas abstrações vazarem. Encapsulamento é fundamental.

9.4 God Class

Classes altamente acopladas são tão problemáticas que discutimos um capítulo inteiro sobre isso. Uma **god class** é aquela classe que controla muitos outros objetos do sistema. Classes assim tendem a crescer mais do que deveriam e passam a "fazer tudo". Os problemas de classes como essas, altamente acopladas, também já são conhecidos. Uma dependência pode forçar mudanças na classe que a consome, e quando uma classe depende de outras 30, isso a faz frágil.



Gerenciar as dependências é a parte mais difícil em um sistema OO. Na imagem, veja uma classe bastante acoplada. `B`, `C`, `D` ou `E` podem propagar mudanças para a classe `A`. A ideia é você usar de tudo o que aprendeu aqui, e encontrar maneiras de diminuir esse acoplamento, seja pensando em abstrações, ou mesmo dividindo as responsabilidades em classes com responsabilidades menores.

9.5 Divergent Changes

Divergent changes é o nome do mau cheiro para quando a classe não é coesa, e sofre alterações constantes, devido às suas diversas responsabilidades. Mais uma vez, classes não coesas possuem baixo reúso, apresentam mais bugs e são mais complexas do que deveriam.

Esse exemplo é bastante forçado, mas é suficiente para você perceber que uma classe que tem muitas responsabilidades tem muitas razões para mudar. Ela também provavelmente possui muitas dependências, e reutilizá-la não é fácil. Divida as responsabilidades o máximo que puder.

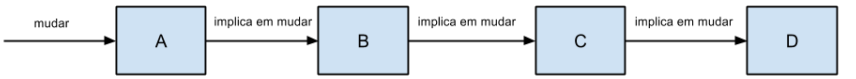
Use tudo o que aprendeu aqui para criar classes coesas.

9.6 Shotgun Surgery

Sabe quando seu usuário pede uma mudança no sistema e, para que isso aconteça, você precisa modificar 20 arquivos de uma só vez? Esse mau cheiro é conhecido por **shotgun surgery**, e é comum em códigos procedurais.

Isso é geralmente típico de sistemas cujas abstrações foram mal boladas. Nesses casos, como a abstração não é suficiente, desenvolvedores acabam implementando pequenos trechos de código relacionados em diferentes lugares. Isso provoca a alteração em cascata.

Lembre-se que além de você precisar alterar em diversos pontos, provavelmente você precisará encontrar esses pontos. Ou seja, você apelará para algum tipo de `grep` ou `CTRL+F`, e com certeza deixará algum deles passar.



Novamente, um sistema OO é um quebra-cabeça. Você precisa pensar muito bem no encaixe das suas peças.

9.7 Entre outros

A lista de maus cheiros é grande. Aqui me preocupei em listar os que acredito terem mais relação com os princípios e práticas discutidos neste livro. Fica a dica: além de estudar e ler sobre boas práticas, leia também sobre maus cheiros. É bom conhecer o outro lado da história.

Capítulo 10

Métricas de código

Se você pensar bem, este livro é até um pouco triste. Ele mostrou diversos problemas que encontramos em códigos do mundo real. Acho que isso deixa bem claro o quanto fazer software é complicado. Talvez até mais complicado que outras engenharias, dado que um software evolui mais rápido do que um prédio ou um carro.

Saber escrever código de qualidade não quer dizer que sempre o escreveremos dessa forma. Criar um bom projeto de classes demanda tempo e muitas rodadas de revisão e feedback. A grande questão é: **como fazer para detectar possíveis problemas em nosso design antes que eles se tornem problemas de verdade?**.

É impossível dizer, com 100% de certeza, se uma classe tem problemas de coesão ou de acoplamento. Humanos não conseguem fazer isso e máquinas também não. Mas por que não criarmos heurísticas? Elas poderiam nos

apontar para um pequeno conjunto, dentre as milhares de classes do sistema, que possivelmente contém problemas.

Por exemplo, é difícil dizer o tamanho certo que um método deve ter. Mas sabemos que um método de 200 linhas provavelmente pode ser melhorado. Não sabemos se uma classe deve estar acoplada a no máximo 5 ou 6 outras classes, mas sabemos que, se ela depende de outras 30, isso pode ser um problema.

Essas **métricas de código** podem nos ser bastante úteis para identificar possíveis trechos problemáticos. Neste capítulo, darei uma visão superficial sobre algumas métricas de código conhecidas, e como elas podem nos ajudar no dia a dia.

10.1 Complexidade ciclomática

Quando um método é complexo? Geralmente quando ele tem muita linha de código ou quando ele tem muitos possíveis diferentes caminhos a serem executados. Veja, por exemplo, o código:

```
public int conta(int a, int b) {
    int total = 0;
    if(a>10) total += a+b;
    if(b>20) total+= a*2+b;
    return total;
}
```

Se você olhar com atenção, verá que esse método tem 4 possíveis caminhos a serem executados:

1) quando ambas as condições são verdadeiras;

2) quando ambas são falsas;

3) quando somente a primeira é verdadeira;

4) quando somente a segunda a verdadeira.

Ou seja, quanto mais controladores de fluxo em um código, como `if`s, `for`s e `while`s, mais caminhos esse código terá e, por consequência, se tornará mais complicado de ser entendido.

Esse número 4 é o que conhecemos por **complexidade ciclomática** desse método. A ideia é que quanto maior esse número, mais complexo aquele método é. Esse número também é conhecido por *Número de McCabe*, pois ele fez estudos sobre o assunto, e mostrou que uma maneira simples e efetiva de calcular esse número é contar a quantidade de instruções de desvio existentes e adicionar 1 ao final. Ou seja, o número de McCabe para o método acima seria 3 (temos 2 `if`s + 1).

Métricas de código podem ser calculadas em diferentes níveis. Aqui, calculamos o número de McCabe em nível de método. Mas podemos generalizá-la para nível de classe. Para isso, podemos somar a complexidade ciclomática de cada um dos métodos. Se a classe `NotaFiscal` tem os métodos `calculaImposto()` e `encerra()`, cujas complexidades são 3 e 5, respectivamente, podemos dizer que a complexidade total da classe é 8 (5+3).

Qual um bom número para complexidade ciclomática?

Como sempre, não há um número mágico, afinal é difícil dizer o tamanho certo para um método. Alguns pesquisadores tentaram encontrar esse número ideal, aliás. Na maioria dos estudos, eles pegaram milhares de projetos de código aberto, e viram como esses números se distribuíam. Meu grande problema com esse tipo de estudo é que eles tendem a não levar em consideração o domínio do projeto, nem o contexto daquela classe.

Mais para frente, mostrarei como faço para avaliar os números que encontro. Por enquanto, perceba que não há um número ideal.

10.2 Tamanho de métodos

Tamanho é algo pode nos dar um bom feedback. Sempre que pensamos em tamanho de coisas, pensamos em linhas de código. Métodos muito grandes podem ser problemáticos, sim. Portanto, monitorar linhas de código pode ser de grande valia.

Mas podemos olhar para quantidade de outras coisas, como quantidade de atributos em uma classe. Uma classe com 70 atributos provavelmente é

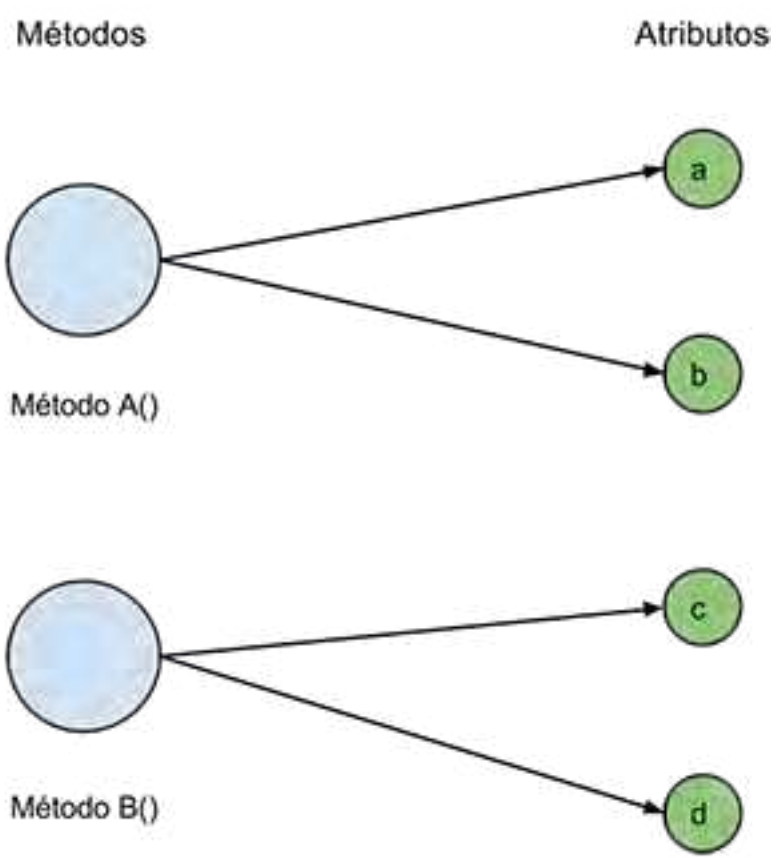
mais difícil de ser mentida. Por que também não olhar a quantidade de métodos em uma classe? Uma classe com 80 métodos provavelmente é difícil de ser mantida e faz coisa demais, afinal ela tem 80 comportamentos diferentes.

O número de variáveis declaradas dentro de um método também pode nos dar alguma noção de complexidade. Afinal, se o método precisa de muitas variáveis, é porque ele é complicado. A quantidade de parâmetros que um método recebe, ou até mesmo o tamanho da árvore de herança, também nos diz um pouco sobre aquela classe.

10.3 Coesão e a LCOM

Discutimos um capítulo inteiro sobre como criar classes coesas. Mas como fazer para detectar o problema? Afinal, coesão é algo totalmente dependente de contexto. É difícil, e o máximo que conseguimos fazer é "chutar" o que é uma responsabilidade dentro da classe.

Sabemos que uma classe é um conjunto de atributos e comportamentos que manipulam esses atributos. Se a classe for coesa, esses comportamentos manipulam boa parte desses atributos, afinal todos eles dizem respeito à mesma responsabilidade. Se a classe não for coesa, ela provavelmente contém um conjunto de atributos que são manipulados apenas por alguns métodos, e outro conjunto de atributos que são manipulados apenas por outros métodos. Ou seja, é como se a classe estivesse dividida em 2, dentro dela mesma. A figura a seguir deixa mais claro. Veja que o método `A()` manipula dois atributos, e o método `B()` manipula outros dois. Ou seja, talvez essa classe tenha duas responsabilidades.



É isso que a métrica conhecida por **LCOM** (*Lack of Cohesion of Methods*) contabiliza. Em alto nível, ela conta o número desses diferentes conjuntos de responsabilidades dentro de uma classe. Como o próprio nome diz, ela mede a falta de coesão de uma classe, ou seja, quanto maior esse número, menos coesa a classe é.

Existem várias versões dessa métrica. Todas elas tentando contornar possíveis problemas, e a mais aceita hoje pela comunidade é a LCOM HS (de Handerson e Sellers, os dois pesquisadores que a modificaram).

Entretanto, a LCOM de maneira geral é bastante criticada pela academia,

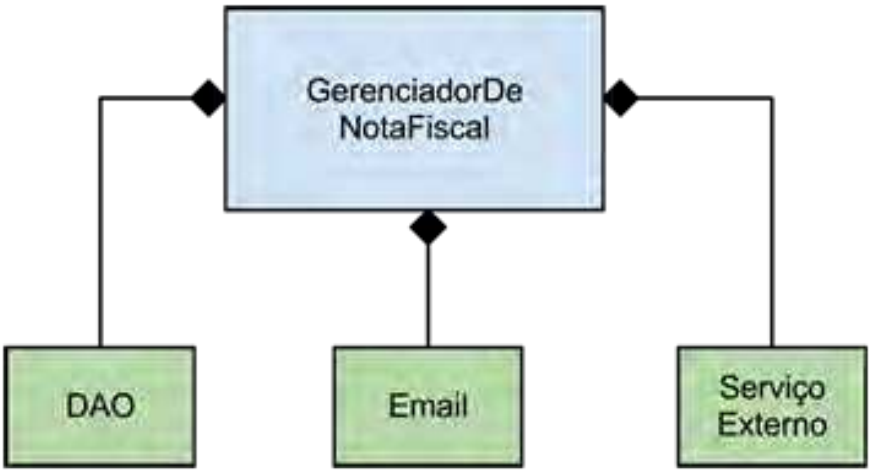
dada a sua sensibilidade. É fácil ela dizer que uma classe não é coesa, mesmo sendo. Um problema típico são aplicações Java. Sabemos que classes Java contêm *getters* e *setters*, e esses métodos manipulam apenas um atributo, fazendo com que esse número cresça.

> **Então não devo usar LCOM?**
>
> Apesar de a métrica ter problemas, lembre-se que ela é apenas uma heurística. Ela pode errar de vez em quando, mas provavelmente a classe que ela mostrar que é realmente pouco coesa pode ser mesmo problemática.
>
> Pense nessas métricas como um filtro. Você não consegue olhar os 50 mil métodos que existem em seu código, mas consegue olhar um a um os 100 que ela filtrar.

10.4 Acoplamento aferente e eferente

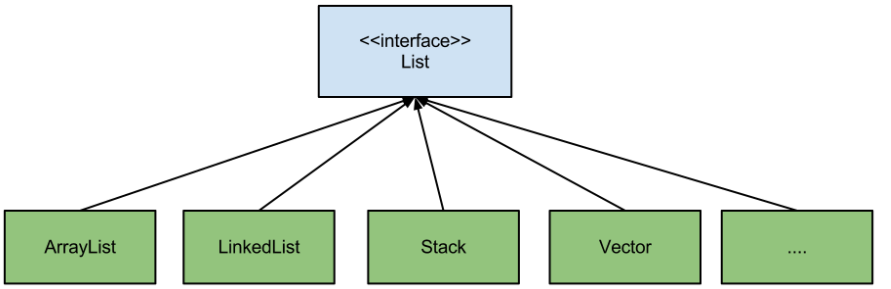
Esses dois termos já foram discutidos no capítulo sobre acoplamento; basta agora darmos os nomes corretos. Quando uma classe depende de diversas outras classes, dizemos que esse é o **acoplamento eferente**. Quanto maior o acoplamento eferente de uma classe, mais ela depende de outras classes e, por consequência, mais frágil ela é.

A figura a seguir exemplifica a classe `GerenciadorDeNotaFiscal`, que depende de classes que falam com o banco de dados, que enviam e-mails e que comunicam com um serviço externo. O acoplamento eferente dela é 3. Novamente, é difícil dizer se 3 é um bom ou mau número, mas, com certeza, se esse número fosse 20, precisaríamos nos preocupar com ela.



O outro lado do acoplamento é o que chamamos de **acoplamento aferente**. Ele mede quantas classes dependem da classe principal. É esse o acoplamento com que nos preocupamos quando queremos ver se a classe é estável ou não. Afinal, se ela é importante para o resto do sistema, provavelmente você não a alterará com frequência; ela é estável.

Na figura, dou o exemplo da interface `List` do Java. Ela não depende de ninguém (ou seja, acoplamento eferente é zero), mas muitas outras classes dependem e a implementam. Ou seja, o acoplamento aferente é alto.



Veja que ambos os acoplamentos são importantes e nos dão informações diferentes sobre a classe. Encontrar uma classe cujo acoplamento aferente é alto, assim como sua complexidade ciclomática, nos dá indício de uma classe muito reutilizada ao longo do código, mas que é complicada. Talvez ela mereça ser refatorada.

10.5 Má nomenclatura

Em nosso dia a dia, lemos muito mais código do que escrevemos. É portanto fundamental que ele seja fácil de ser compreendido. Dar bons nomes às variáveis, métodos e classes é importante. Apesar de nomes ser algo totalmente contextual e difícil de ser entendido por uma máquina, algumas regras básicas podem ser validadas pela máquina.

Variáveis cujo nome contenha muitos caracteres são complicadas de serem lidas. O mesmo para variáveis de uma única letra (a única exceção é, claro, para os `i`s e `j`s, que usamos em nossos `for`s).

A falta de uso de padrões também pode ser algo problemático. Por exemplo, usar *underscore* em linguagens cuja padrão é *camel case*, ou constantes com nomes em minúsculo. Ou qualquer outra convenção que a equipe deve seguir e que, por algum motivo, aquele trecho de código não segue.

10.6 Como avaliar os números encontrados?

Agora que você já conhece diversas métricas de código, é hora de analisar esses dados e com isso descobrir quais são as classes que merecem atenção. Já sabemos que o grande problema é descobrir se o número informado pela métrica é realmente um problema. Qual o limite para dizermos que a classe tem problemas de acoplamento ou de complexidade?

Existem algumas abordagens para acharmos esse limite:

- Você pode usar algum número mágico. Apesar da minha crítica a eles, é uma possível maneira. Na literatura, você encontra números ideais para essas métricas. Basta você compará-los com os seus.
- Você pode criar seu número mágico. Uma ideia um pouco melhor

> talvez seja ter seu próprio número. Você pode usar um pouco de estatística, olhar a distribuição, medir quartis, e descobrir qual o melhor número de cada uma dessas métricas. E, a partir daí, comparar com esse número. Nessa situação, você não está comparando com um número ideal geral, mas sim com aquilo que você já está acostumado a manter.

Você pode também pensar em avaliar seu sistema como um todo, em um nível mais alto que o nível de classes, agrupando os números calculados para as classes. Se você perceber que 90% das classes estão dentro dos limites aceitáveis, você pode dizer que esse sistema é ótimo. Se apenas 70% das classes estão dentro desse limite, ele não é ótimo, é apenas bom. E assim por diante. As porcentagens, obviamente, foram simplesmente um chute (você precisa encontrar os seus), mas empresas com muitos projetos diferentes podem utilizar essa abordagem para comparar a qualidade entre eles.

Lembre-se que a ideia de tudo isso não é dizer se uma classe ou um sistema tem ou não problemas, com 100% de certeza. O objetivo é termos algo que nos ajude a encontrar esse problema de maneira rápida e barata. Pense nelas como um filtro. Você não pode revisar o código seu sistema inteiro o tempo todo, pois ele é grande; a métrica pode, e faz isso rápido. Mesmo que ela acerte em apenas 80% dos casos, se ela selecionou pra você 50 classes do seu sistema, você pode pagar o custo de revisar essas, uma a uma.

10.7 Ferramentas

Infelizmente, ferramentas de métrica de código ainda estão longe de serem boas. Elas raramente calculam tudo o que queremos no momento, e são bem difíceis de serem customizadas. Fazer a ferramenta que calcula LCOM ignorar *getters* e *setters*, por exemplo, pode não ser fácil.

As mais comuns hoje são o Sonar, um plugin para integração contínua, que calcula todas as métricas mencionadas, e mostra a evolução delas ao longo do tempo. Como ela é disparada automaticamente, cada vez que o código é comitado, o desenvolvedor não precisa lembrar de rodar a ferramenta.

Outra bastante famosa é o Eclipse Metrics. Ela é um plugin para o Eclipse que também calcula muitas das métricas citadas. Como ela roda na máquina

do desenvolvedor, ele precisa lembrar de executar a ferramenta; o que, na prática, não acontece. Ela é uma excelente ferramenta para visualizações pontuais ou mesmo para aprendizado.

Uma rápida busca no Google e você também encontrará outras ferramentas pontuais, como JDepend/NDepend, JavaNCSS etc. Todas elas fazem bem seu trabalho.

Lembre-se que a ferramenta não importa, mas sim como você analisará os dados gerados por ela.

Capítulo 11

Exemplo prático: MetricMiner

MetricMiner é uma ferramenta desenvolvida pelo nosso grupo de pesquisa de métricas de código dentro da Universidade de São Paulo. Sua ideia básica é minerar informações que estão salvas dentro de um repositório de código-fonte, como Git e SVN. Ele possibilita ao pesquisador navegar por todas as alterações feitas ao longo do tempo de um projeto, bem como códigos-fontes, nomes de autores, descrições de commits etc. Essas informações são fonte de dados para a grande área de pesquisa em engenharia de software conhecida como *mineração de repositório de código*.

A ferramenta precisa ser flexível, pois é impossível prever o tipo de informação que o pesquisador vai querer. Alguns podem estar interessados na quantidade de commits feitos por cada desenvolvedor; outros podem querer saber com qual frequência determinado arquivo é alterado; outros olham para a quantidade de linhas de código de cada arquivo e se elas crescem ou diminuem ao longo do tempo. O número de perguntas é ilimitado. Navegar pelos

commits deve ser fácil.

A ferramenta possui código aberto e está disponível em http://www.github.com/metricminer-msr/metricminer. Neste capítulo, discutirei as principais decisões de design que tomamos. Elas podem não mais refletir a realidade (afinal, continuamos aprendendo e melhorando nosso código), mas ainda assim as decisões do momento atual são bastante interessantes.

Neste capítulo, preste muita atenção a cada decisão tomada. Olhe cada classe e pense se ela está coesa, como ela está acoplada, se as abstrações são coesas e estáveis. O código, em particular, é o menos importante aqui.

11.1 O projeto de classes do ponto de vista do usuário

O MetricMiner nasceu com a intenção de ser um framework, ou seja, os pesquisadores que quiserem usar a ferramenta deverão implementar suas interfaces, e o framework se encarregará de executá-las no contexto correto.

Todo estudo executado pelo MetricMiner deve implementar a interface `Study`. Ela é bastante simples, e contém apenas um método:

```
public interface Study {
    void execute(MMOptions opts);
}
```

O método `execute()` é o que será invocado, e o MetricMiner passará para ele os parâmetros de configuração que o usuário definiu pela linha de comando. Parâmetros como diretórios dos projetos a serem minerados, quantidade de *threads* que devem rodar simultaneamente, e arquivo de saída.

Implementações de estudos fazem uso da classe `SourceCodeRepositoryNavigator`. Essa classe é a responsável por executar as ações desejadas pelo usuário em cada commit do projeto. Para que ela funcione corretamente, ela precisa conhecer:

- a implementação do repositório de código, afinal ele pode ser Git, SVN, ou qualquer outro;
- as ações que serão executadas em cada commit.

Portanto, os métodos mais importantes dessa classe são: `in(SCMRepository... scm)`, que recebe os repositórios a serem navegados; o método `process(CommitVisitor visitor, PersistenceMechanism writer)`, que recebe um visitante de commits e a maneira como a saída deve ser armazenada; e o método `start()`, que inicia todo o processo de navegação nos commits.

Para representar os repositórios de código, temos a classe `SCMRepository`, que é basicamente uma classe de domínio, com informações como o path físico, o primeiro commit e o último commit. Além disso, ela também contém um `SCM`, que é o motor responsável por interagir com aquele repositório. Para instanciar esses repositórios, existem *factory methods* que fazem o trabalho. Para instanciarmos um repositório Git, por exemplo, fazemos:

```
SCMRepository repo = GitRepository.build("/path/do/projeto");
```

O `CommitVisitor` possui uma interface também simples. Ele possui um método chamado `process`, que recebe o repositório que está sendo processado, um commit, e um mecanismo de persistência. O MetricMiner invocará esse método diversas vezes: uma para cada commit do projeto.

```
public interface CommitVisitor {
    void process(SCMRepository repo, Commit commit,
        PersistenceMechanism writer);
    String name();
}
```

A parte mais trabalhosa para o pesquisador é justamente implementar esses visitantes dos commits. É ali que ele colocará as regras de negócio da sua pesquisa. Se quisermos escrever um simples visitante que conta a quantidade de classes que foram deletadas naquele commit, e imprimir em um arquivo com o nome do projeto, a *hash* do commit e a quantidade, podemos escrever o seguinte código:

```
public class ContadorDeClassesRemovidas implements
CommitVisitor {
```

```
    @Override
    public void process(SCMRepository repo,
        Commit commit,
        PersistenceMechanism writer) {

        int qtdDeDelecoes = 0;
        for(Modification m : commit.getModifications()) {
            if(m.getType().equals(ModificationType.DELETE)) {
                qtdDeDelecoes++;
            }
        }

        writer.write(
            repo.getLastDir(),
            commit.getHash(),
            qtdDeDelecoes);
    }
}
```

O mecanismo de persistência é bastante simples. Ele possui um método `write` que recebe uma lista de strings, e essas strings serão salvas em algum lugar. A implementação mais convencional é o arquivo `CSV`, pois é um formato fácil de ser lido por ferramentas de estatística, como a linguagem R.

```
public interface PersistenceMechanism {
    void write(Object... line);
    void close();
}
```

Com os visitantes escritos, podemos finalmente escrever nosso estudo. O exemplo a seguir é um estudo que analisa o comportamento de uso das APIs do Selenium, um framework de testes de sistema, em baterias de teste. Repare que ele usa um repositório Git e possui dois visitantes de commit.

```
public class SeleniumStudy implements Study {

    @Override
    public void execute(MMOptions opts) {
```

```
        new SourceCodeRepositoryNavigator(opts)
            .in(GitRepository.build("/path/do/projeto"))
            .process(new AddsAndRemoves(),
                new CSVFile("addsremoves.csv"))
            .process(new CommittedTogether(),
                new CSVFile("committed.csv"))
            .start();

    }
}
```

Resumindo, do ponto de vista do pesquisador, ele precisa escrever uma implementação de `Study`, que é a classe que será executada pelo MetricMiner, e implementar diversos visitantes de commits. O MetricMiner se encarregará de navegar naquele repositório de código e invocar os visitantes para cada commit encontrado. A saída também será salva de acordo com o mecanismo de persistência passado.

Apesar da quantidade de classes que ele precisa conhecer, é tudo bastante simples para o pesquisador. Veja que o usuário do framework não precisa dos detalhes internos da ferramenta.

11.2 O projeto de classes do ponto de vista do desenvolvedor

Do ponto de vista interno, o MetricMiner é um pouco mais complexo, afinal, ele precisa fazer com que o estudo seja executado e os visitantes sejam invocados na hora certa.

As implementações concretas de `SCM`, por exemplo, são as classes que efetivamente comunicam-se com o sistema controlador de código. Essas implementações não são simples, afinal, interagir diretamente com essas ferramentas não é fácil. A classe `Git`, por exemplo, faz uso de `JGit`, um projeto de código aberto que facilita essa comunicação.

Essa interface possui dois métodos: um que nos retorna a lista de *changesets*, e outro que retorna um commit detalhado. Ou seja, se um desenvolvedor quiser dar suporte a um novo gerenciador de código, ele precisa apenas de-

volver a lista de *hash* dos commits (que aqui chamamos de *changesets*) e os detalhes de commit em particular.

```
public interface SCM {
    List<ChangeSet> getChangeSets();
    Commit getCommit(String id);
}
```

A classe `Commit` é bastante complexa. Ela contém as informações de alto nível de um commit, como data, hora, autor, mensagem, e uma lista de modificações. Cada modificação é uma alteração feita naquele commit. Um arquivo alterado, por exemplo, é uma modificação do tipo *MODIFY*. Ela também contém o `diff`, ou seja, as linhas modificadas naquele momento:

```
public class Commit {

    private String hash;
    private Committer committer;
    private String msg;
    private List<Modification> modifications;
    private String parent;
    private Calendar date;

    // getters
}

public class Modification {

    private String oldPath;
    private String newPath;
    private ModificationType type;
    private String diff;
    private String sourceCode;

    // getters

}
```

Já o método `start()` do `SourceCodeRepositoryNavigator` é o que faz a mágica acontecer. Ela divide a quantidade de commits a serem pro-

cessados em diferentes listas (uma para cada *thread* que será executada) e dispara uma *thread* para cada conjunto de commits. Cada *thread*, por sua vez, itera neles e dispara todos os visitantes.

Como calcular métricas de código é algo comum, um visitante de commit que já vem com o framework é o `ClassLevelMetricCalculator`. Ele é bastante simples: ele executa uma métrica de código para cada modificação existente. A ferramenta já vem com diversas métricas de código, e o pesquisador pode fazer uso delas. A calculadora de métricas recebe a métrica que será executada em seu construtor. Mas não a métrica diretamente, e sim a fábrica. Algumas métricas podem ser mais complicadas de serem criadas, portanto, uma fábrica é necessária.

Veja o código a seguir, onde um pesquisador passa a calculadora de métrica de código, bem como a fábrica que instancia a métrica concreta:

```
.process(new ClassLevelMetricCalculator(
                new LackOfCohesionFactory()),
    new CSVFile("lcom.csv"))
```

A fábrica também é necessária por uma outra razão. Métricas de código geralmente são implementadas por meio de navegações em árvores abstratas de sintaxe (as famosas *ASTs*) e muitas dessas navegações guardam estados em seus objetos. Como essas métricas são executadas diversas vezes para diversas classes, o estado gerado na classe anterior afetaria a próxima classe. Ou seja, precisamos de novas instâncias da métrica o tempo inteiro. Para isso, nada melhor do que ter uma classe que sabe instanciar a métrica a hora que precisar.

Uma métrica de código, para se encaixar no desenho atual, precisa seguir a seguinte convenção. Ter uma fábrica que implementa `ClassLevelMetricFactory`. Essa fábrica nos devolve a instância concreta da métrica, bem como seu nome. E também uma classe que implementa `ClassLevelMetric` é a métrica de código em si e nos devolve um número com o valor daquela métrica para o código-fonte passado como string.

```
public interface ClassLevelMetricFactory {

    ClassLevelMetric build();
    String getName();
```

```
}

public interface ClassLevelMetric extends CodeMetric {

    double calculate(String sourceCode);

}
```

De maneira análoga, temos a `MethodLevelMetric`, que são métricas calculadas no nível de método, e não no nível de classe. Nesse caso, o retorno não é simplesmente um número, mas sim um mapa com o nome do método e seu valor associado.

11.3 Conclusão

Por fim, repare que o framework tem algumas decisões e pontos de flexibilização no projeto:

- Gerenciadores de código devem ser implementações de `SCM`.
- Essas implementações possuem fábricas estáticas para se gerar `SCMRepository`, a classe de domínio que representa o projeto, específicas.
- Métricas de código devem ter fábricas específicas e ser implementações de `ClassLevelMetric`.
- Estudos devem ser implementações de `Study`, e eles são invocados automaticamente pelo framework.
- Visitantes de commit devem ser implementações de `CommitVisitor`, e o framework invocará o visitante para cada commit naquele repositório.

Se você observar a distribuição de pacotes, verá que eles deixam bem claro essa subdivisão. Repare que todos eles possuem subpacotes, mas aqui, podemos tratá-los em alto nível:

- O pacote `br.com.metricminer2`, pacote raiz, possui o executável do framework.
- O pacote `br.com.metricminer2.domain` contém todas as classes de domínio, como `Commit`, `Modification` etc.
- O pacote `br.com.metricminer2.scm` contém as implementações de SCM, bem como a interface `CommitVisitor`.
- O pacote `br.com.metricminer2.persistence` contém os mecanismos de persistência.
- O pacote `br.com.metricminer2.metric` contém as métricas de código.

Agora que você tem uma explicação de alto nível sobre o projeto, sugiro a você navegar no código, e entender melhor como elas funcionam. Entenda os pontos de flexibilização dados e como as classes se conectam.

Capítulo 12

Conclusão

Fazer um bom projeto de classes orientado a objeto não é fácil. É muita coisa para pensar. É pensar em ter classes com responsabilidades bem definidas e, assim, ganhar em coesão. É pensar em ter classes com dependências bem pensadas e estáveis, e dessa forma ganhar em flexibilidade. É pensar em esconder os detalhes de implementação dentro de cada classe, e ganhar a propagação boa de mudança.

Fazer tudo isso requer prática e experiência. Lembre-se também que você não criará um bom projeto de classes de primeira; valide, aprenda e melhore o seu projeto de classes atual o tempo todo.

12.1 Onde posso ler mais sobre isso?

Sem dúvida alguma, há muito material gratuito na internet sobre o assunto. Se você digitar o nome de qualquer um dos princípios explicados, vai encontrá-

los facilmente. Mas não posso deixar de citar os livros que me influenciaram demais ao longo da minha carreira e que, com certeza, recomendo a todos:

- *Agile Principles, Patterns and Pratices*, do Robert Martin. Um livro bastante extenso, onde ele discute os princípios SOLID com um alto grau de profundidade.
- *Growing Object-Oriented Software, Guided by Testes*, do Steve Freeman e Nat Pryce. Um livro que me mostrou muito sobre como fazer TDD e obter feedback em relação à qualidade do código. Além disso, as discussões sobre design são bastante interessantes.
- *Design Patterns*, da Gang of Four. Estudar a fundo cada um dos padrões de projeto do GoF foi o que realmente me fez entender orientação a objetos. Lá você encontra um catálogo imenso de orientação a objetos bem aplicada.
- *Refactoring to Patterns*, do Joshua Kerievsky. O livro discute como transformar códigos procedurais em códigos OO. Ajudou-me muito a entender os reais problemas do mundo procedural e por que OO é tão importante.

Não posso também deixar de mencionar blogs e entrevistas de outros grandes pensadores da engenharia de software como Martin Fowler, Erich Gamma, Ralph Johnson, David Parnas, Joe Yoder etc.

Não deixe de me seguir no Twitter (*@mauricioaniche*), ou mesmo participar da lista de discussão do grupo: https://groups.google.com/forum/#!forum/oo-para-ninjas.

12.2 Obrigado!

Espero que este livro tenha lhe ensinado bastante coisa, e que você esteja apto a pôr tudo isso em prática.

Um forte abraço!